8-2016

Effective and Economical Content Delivery and Storage Strategies for Cloud Systems

Yuhua Lin
Clemson University, yuhual@clemson.edu

Effective and Economical Content Delivery and Storage Strategies for Cloud Systems

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

---

by
Yuhua Lin
August 2016

Accepted by:
Professor Haiying Shen, Committee Chair
Professor Richard R. Brooks
Professor Adam Hoover
Professor James (Zijun) Wang

Abstract

Cloud computing has proved to be an effective infrastructure to host various applications and provide reliable and stable services. Content delivery and storage are two main services provided by the cloud. A high-performance cloud can reduce the cost of both cloud providers and customers, while providing high application performance to cloud clients. Thus, the performance of such cloud-based services is closely related to three issues. First, when delivering contents from the cloud to users or transferring contents between cloud datacenters, it is important to reduce the payment costs and transmission time. Second, when transferring contents between cloud datacenters, it is important to reduce the payment costs to the internet service providers (ISPs). Third, when storing contents in the datacenters, it is crucial to reduce the file read latency and power consumption of the datacenters. In this dissertation, we study how to effectively deliver and store contents on the cloud, with a focus on cloud gaming and video streaming services. In particular, we aim to address three problems. i) Cost-efficient cloud computing system to support thin-client Massively Multiplayer Online Game (MMOG): how to achieve high Quality of Service (QoS) in cloud gaming and reduce the cloud bandwidth consumption; ii) Cost-efficient inter-datacenter video scheduling: how to reduce the bandwidth payment cost by fully utilizing link bandwidth when cloud providers transfer videos between datacenters; iii) Energy-efficient adaptive file replication: how to adapt to time-varying file popularities to achieve a good tradeoff between data availability and efficiency, as well as reduce the power consumption of the datacenters.

In this dissertation, we propose methods to solve each of aforementioned challenges on the cloud. As a result, we build a cloud system that has a cost-efficient system to support cloud clients, an inter-datacenter video scheduling algorithm for video transmission on the cloud and an adaptive file replication algorithm for cloud storage system. As a result, the cloud system not only benefits the cloud providers in reducing the cloud cost, but also benefits the cloud customers in reducing

their payment cost and improving high cloud application performance (i.e., user experience).

Finally, we conducted extensive experiments on many testbeds, including PeerSim, PlanetLab, EC2 and a real-world cluster, which demonstrate the efficiency and effectiveness of our proposed methods. In our future work, we will further study how to further improve user experience in receiving contents and reduce the cost due to content transfer.

Acknowledgments

I would like to express my sincere appreciation and thanks to my advisor Dr. Haiying Shen, whose passion and earnest manner in research have transformed me and will benefit me for the rest of my life. Her guidance helped me overcome numerous difficulties and finish my Ph.D study.

I would also like to thank my committee members: Dr. Richard R. Brooks, Dr. Adam Hoover, and Dr. James (Zijun) Wang, for their valuable comments and suggestions.

I thank my colleagues in the Pervasive Communications Laboratory: Ze Li, Kang Chen, Guoxin Liu, Chenxi Qiu, Bo Wu, Liuhua Chen, Zhuozhao Li, Jinwei Liu, Li Yan, Ankur Sarker, Haoyu Wang, for helping me on my research and also daily life. You guys are wonderful companies, and together we went through all good times and hard times.

I especially thank my beloved parents, Yashui Lin and Yinying Liao, for their constant support and unconditional love. I will forever be thankful to all my families who are the sources of all my happiness.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cloud computing has proved to be an effective infrastructure to host various applications and provide reliable and stable services. Content delivery and storage are two main services provided by the cloud such as online gaming [61, 62] and video streaming services [6, 60, 104]. Cloud providers (e.g., Amazon) offer various pay-as-you-use cloud based services (e.g., Amazon Web Services) to cloud customers (e.g., Netflix and Hulu) [15, 89]. A high-performance cloud can reduce the cost of both cloud providers and customers, while providing high application performance to cloud clients. In this dissertation, we aim to develop a high-performance cloud for the content delivery and storage service.

In these cloud-based services, several issues need to be studied in order to maintain stable and superior performance as shown in Figure 1.1. The figure shows some key issues in cloud computing, i.e., content delivery and storage, while the content delivery issue can be further divided into delivery from the cloud to users and delivery inside the cloud. First of all, as the cloud needs to deliver a large amount of contents to the users continuously, we need to reduce cloud bandwidth costs for the cloud-based service providers and reduce the transmission time. Second, as the cloud providers store user data in datacenters that are in different geographical areas and frequently transfer files between two datacenters to replicate and backup user data, it is important to reduce the bandwidth payment costs for these inter-datacenter data transmissions. Third, the cloud providers store user data in datacenters, which is a cluster of commodity servers. Cloud-based services generate frequent and intensive file read operations towards files stored in the datacenter, so we need to reduce the file read latency and power consumption of datacenters. To sum up, in this dissertation, we aim to

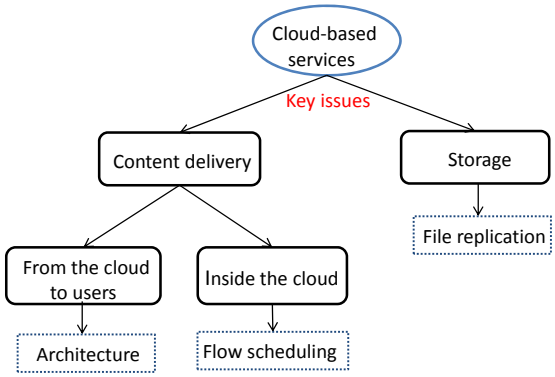


Figure 1.1: Overview of the key issues in cloud computing and the proposed solutions in this dissertation

study how to improve the effectiveness of content delivery and storage on the cloud. Specifically, for the first issue, we aim to facilitate the content delivery from the cloud to users by extending the cloud architecture with supernodes. For the second issue, we aim to reduce the bandwidth costs by developing a cost-efficient inter-datacenter video scheduling algorithm. For the third issue, we design an adaptive file replication system reduce the file read latency and power consumption of datacenters.

In the following, we present the importance of each of the issues to enhance the cloud performance and the motivation for us to address the issues.

Cloud computing to support Massively Multiplayer Online Game (MMOG). With the increasing popularity of MMOG and fast growth of mobile gaming, thin-client MMOG will become a new gaming model. To support thin-client MMOG, game service providers begin to deploy their gaming service on the cloud, which is called cloud gaming. Cloud gaming exhibits great promises over the conventional MMOG gaming model as it frees players from the requirement of hardware and game installation on their local computers. However, as the graphics rendering is offloaded to the cloud, the data transmission between the end-users and the cloud significantly increases the response latency and limits the user coverage, thus preventing cloud gaming to achieve high QoE. To solve this problem, previous research suggested deploying more datacenters, but it comes at a prohibitive cost.

Inter-datacenter video scheduling. As a large collection of videos are stored and transferred between datacenters in the cloud, cloud providers are charged by ISPs for inter-datacenter transfers under the dominant percentile-based charging models. In order to minimize the payment costs for cloud providers, existing works aim to keep the traffic on each link under the charging

volume (i.e., 95th percentile traffic volume from the beginning of a charging period up to current time). However, these methods cannot fully utilize each link's available bandwidth capacity, and may increase the charging volumes. Therefore, the cloud providers need a new inter-datacenter video scheduling algorithm to minimize the payment costs.

Adaptive file replication. Besides that, cloud providers store files in datacenters, where a large amount of files are stored, processed and transferred simultaneously. To increase the file availability, some cloud storage systems create and store three replicas for each file in randomly selected servers across different racks. However, they neglect the file heterogeneity and server heterogeneity, which can be leveraged to further enhance data availability and cloud storage system efficiency. As file have heterogeneous popularities, a rigid number of three replicas may not provide immediate response to an excessive number of read requests to hot files, and waste resources (including energy) for replicas of cold files that have few read requests. Also, servers are heterogeneous in network bandwidth, hardware configuration and capacity (i.e., the maximal number of service requests that can be supported simultaneously), it is crucial to select replica servers to ensure low replication delay and request response delay.

1.1 Problem Statement

In this section, we present the details of each of the three problems addressed in this dissertation: i) cloud gaming to support thin-client MMOG, ii) payment costs for inter-datacenter video transfers, and iii) file replication in cloud storage system.

1.1.1 Cloud Gaming to Support Thin-Client MMOG

Though the advantages of cloud gaming makes it a very promising model to cater to thin-client MMOG, it currently faces severe challenges (i.e., latency, network connection, user coverage and bandwidth cost) that prevent it from becoming a leading gaming model. First, response latency is a critical factor in user quality of experience (QoE). By offloading computation to a remote host, cloud gaming suffers from long *response latency*, which is the delay in sending the user action information and game video between the end-user and the cloud. Second, cloud gaming services post a strict requirements of high-speed network connection for a relatively high constant downlink

bandwidth (e.g., 5Mbit/s recommended by OnLive). Third, the shortage of datacenters limits user coverage. Players begin to notice a response delay of 100ms [63]; 20ms attributed to playout delay on client side and processing delay on the cloud, 80ms attributed to the network latency. The playout delay of a client includes the time to send action information, receive and play the game video. Choy *et al.* [36] found that Amazon's EC2 (with 13 datacenters) can provide a median latency of 80ms or less to only fewer than 70% of their 2500 tested end-users in the US. Existing cloud infrastructure is not sufficient for hosting cloud gaming, as a sizeable portion of the population would experience significantly degraded QoE. Fourth, besides server time, bandwidth costs represent a major expense when renting on-demand resources. An average traffic of 27TB per 12 hours leads to about $130k monthly fee for bandwidth with Amazon EC2's price (i.e., $0.085 per GB) [29]. Considering the MMOG's huge user scale, these costs can significantly affect the feasibility of the cloud computing of thin-client MMOG [26]. In spite of the previous research efforts on cloud gaming, except deploying more datacenters which is costly, no other approaches have been proposed to handle its critical challenges. We propose light-weight strategies to tackle the challenges to support thin-client MMOG.

1.1.2 Payment Costs for Inter-Datacenter Video Transfers

At the same time, a large amount of videos are store in the cloud storage system and accessed by the viewers. The storage and replication of these videos will lead to large volume of inter-datacenter traffic. Many previous studies focus on controlling the new traffic volume below the charging volume [45, 48, 71, 72, 81, 107, 116] in order to minimize the bandwidth payment cost on inter-datacenter video traffic to ISPs. We can mainly classify such previous studies into two groups: *store-and-forward* and *optimal routing path*.

The *store-and-forward* methods [71, 72, 81] take advantage of the spatial and temporal features of the inter-datacenter video traffic. The spatial feature means that at a specific time, datacenters in different geographic areas exhibit different traffic loads and available bandwidth capacities, which highly depend on the user demands from different geographic areas. The temporal feature means that the traffic loads on a datacenter exhibit strong diurnal patterns that are correlated with the local time [50]. Based on these two features, these methods predefine peak and off-peak hours for each datacenter based on its local time and geographic area, and then utilize the leftover traffic volume (which is the charging volume minus the actual traffic volume) during off-peak hours to transfer delay-tolerant data flows. For example, for datacenters on the East Coast, their off-peak

hours are 3-6am EST, and their peak hours are 9-12pm EST. When datacenter i wants to send some delay-tolerant data flows to datacenter j, which however is at its peak hours and there is no overlap between i and j's off-peak hours, datacenter i then sends the data flows to an intermediate datacenter k which is in off-peak hours. k stores these flows temporally and forwards them to j when both k and j are in off-peak hours. However, the *store-and-forward* methods perform the transmissions of delay-tolerant data flows during the predefined off-peak hours. They fail to fully utilize the available bandwidth capacities of the light-traffic links during the peak hours, also, the link's charging volume will increase when a large number of non-delay-tolerant videos are transmitted during the peak hours. Such coarse-grained scheduling of data flows cannot reduce the bandwidth costs as much as possible.

The optimal routing path [45, 48, 107, 116] optimize the routing paths for video flows to minimize the charging volume on each link. As the bandwidth costs of transmitting the same amount of videos vary across different inter-datacenter links, if the transmission of a video is expected to exceed the charging volume on a link, the video will be transferred over an alternating path to maximize the utilization of other links without increasing their charging volumes. However, these methods transmit each video immediately when the video transmission request arrives at the source datacenter regardless of their deadlines. Therefore, these methods can easily reach the charging volume of current link and create many reroute requests when a large number of video transfer requests arrive simultaneously, which many increase the charging volumes of some links. Also, a link's available bandwidth capacity is not fully utilized when the cumulated transmission rates of all currently transmitted videos is less than the link's available bandwidth capacity.

1.1.3 File Replication in Cloud Storage System

The uniform replication policy neglects the file and server heterogeneity, which can be leveraged to further enhance data availability and file system efficiency. First, the files in a large cluster exhibit wide disparity in popularity. For example, the data in HDFS can be classified into four categories according to their access patterns and popularity [32,98]: hot data, cooled data, cold data and normal data. For cold data that is rarely requested, too many replicas may not improve file availability, but instead lead to unnecessary storage cost. Therefore, in order to improve replica efficiency, we should increase the replication factor (i.e., the number of replicas of a file) of hot data to guarantee data availability and load balance, and reduce the replication factor of cold data to

save the storage cost.

Second, energy consumption contributes a significant portion of management cost for datacenters [106]. The energy cost of equipment during its lifetime is comparable to the initial equipment purchase price [30]. Existing file systems randomly select servers in each rack to replicate data (called replica destinations), but do not consider selecting replica destinations to reduce energy consumption. The file replication system actually can reduce energy consumption based on file popularity heterogeneity. We can separate the cluster into hot servers with high CPU utilization (i.e., high power consumption) and cold servers with low CPU utilization (i.e., low power consumption), and place the replicas of popular data in hot servers, which provide high performance, and place the replicas of cold data in cold servers as data backup.

Third, the random selection of replica destinations neglects server heterogeneity (i.e., different servers vary in network capacities and data request handling capacities). The writes due to creating replicas in production clusters at Facebook and Microsoft account for almost half of all cross-rack traffic [33]. Though the network inside clusters is generally underutilized, there exist some bottleneck links resulting from the network usage imbalance [49]. As the traffic in multi-tenant datacenters is not controlled, the traffic congestion of bottleneck links leads to performance degradation inside clusters [24]. If a large number of replicas are written to the same server simultaneously, the server may run out of network capacity and data request handling capacity. Thus, it is important to choose replica destinations to steer replica transfers away from network bottlenecks and overloaded servers.

1.2 Research Approach

According to the discussion of the challenges in Section 1.1, the cloud providers need a cost-efficient gaming system to support thin-client MMOG, an inter-datacenter video scheduling algorithm to save the bandwidth costs and an adaptive file replication solution for effective file storage on the cloud. In this dissertation, we have proposed different algorithms to solve these problems. We briefly describe our solution for each problem below and will present the details in the next section.

1.2.1 Extend Cloud Gaming for Thin-Client MMOG with High Quality of Experience

The great promises of cloud gaming and the obstacles it faces motivate us to explore approaches to efficiently handle the challenges. Though previous study suggested to deploy more datacenters [63], building and maintaining a large number of datacenters is cost-prohibitive. In this dissertation, we propose a lightweight system called *CloudFog*. We introduce a concept called "fog", formed by powerful supernodes, that are close to end-users and connect them with the cloud. Considering that most desktop systems are underutilized in most organizations; they are idle around 95% of the time [70], the supernodes can be from these idle resources or from players' computers. In CloudFog, the intensive computation [10,21] of the new game state of the virtual world is conducted in the cloud. The cloud sends update messages to supernodes, which updates their virtual world, render game videos for different players and streams videos to them. Thus, users without high speed network connection to cloud or out of the coverage of the cloud can be supported by nearby supernodes, and the cloud does not need to transmit entire game videos to far-away users, which increases user coverage, shortens response latency and ensures relatively high-speed network connection for high QoE and reduces bandwidth cost. The difference between EdgeCloud and CloudFog lies in the responsibility of newly added servers. In EdgeCloud, the addition of a small number of servers are used to store and compute game status and render new game videos; while in CloudFog, the storage and computation are carried out on the cloud, servers are only used to render new game videos and stream them back to the players. As the rendering work does not require high hardware configuration, given the same amount of revenue, *CloudFog* can deploy more servers than EdgeCloud by using proper incentives to motivate players or organizations to contribute their spare machines. Specifically, CloudFog incorporates the following strategies to handle the challenges and enhance QoE. Strategies (2) and (3) are based on the phenomenon that different games have different tolerance on packet loss rate and response delay [73].

(1) **Fog-assisted cloud gaming infrastructure.** We leverage the hardware and bandwidth capacity of some idle machines, and deploy them as supernodes. These supernodes constitute the "fog", and are responsible to stream game videos for nearby players.

(2) **Receiver-driven encoding rate adaptation.** In order to ensure the playback continuity even in network congestion, when a supernode streams a game video to a player, it adaptively

changes the encoding rate (hence the quality) of the video based on the segment size in the player's buffer according to the game's tolerance on delay and packet loss.

(3) **Deadline-driven sender buffer scheduling.** To meet response latency requirement of each game, supernodes give higher priority to lower delay-tolerant game videos to send out, and drop different numbers of packets from different game videos based on their packet loss tolerant degree.

(4) **Social network based server assignment.** The communication between servers in a datacenter for generating game videos leads to latency. As social friends always play together [23], we assign social friends who usually play together to the same server, so their interaction will not trigger communication between different servers, thus reducing response latency.

1.2.2 Economical and Deadline-Driven Inter-Datacenter Video Flow Scheduling

We propose an economical and deadline-driven video flow scheduling system, called EcoFlow. It is based on the fact that different video flows have different deadlines. Different applications from cloud customers have different service-level agreements (SLAs) that specify data Get/Put bounded latency [59] or a service probability [8] by ensuring a certain number of replicas in different locations [91]. Thus, cloud providers would like to assign shorter transmission deadlines (deadline in short) to videos in applications with more stringent SLAs in order to minimize the SLA violation penalty to maximize their profits [16, 45]. Different videos in one application also have different deadlines. For example, the flows for new video dissemination to a datacenter to serve user requests should have more stringent deadlines than the flows for video replication backups to boost availability. Based on the different deadlines of video flows, the basic idea of EcoFlow is to postpone the transfers of some delay-tolerant videos while still ensure their transmission within deadlines if the transmission of these videos will increase the current charging volume. The EcoFlow system includes three key steps.

- Step 1: **available bandwidth capacity estimation**. By comparing the charging volume and expected traffic volume, we estimate the available bandwidth capacity on each link, which is the maximum transmission rate that a link can provide in the next time interval without increasing the current bandwidth cost.

- Step 2: **deadline-driven flow scheduling**. Based on the estimated traffic capacity, we sort the flows on each link based on their deadline tightness so that videos with early deadlines have high priorities to finish transmission. Flows that are expected to miss their deadlines are splitted into subflows, which will be rerouted to other under-utilized links in order to meet their deadlines.

- Step 3: **alternating routing path identification**. In order to deliver these subflows by their deadlines, we rely on Dijkstra's algorithm [43] to find the shortest path between the source and the destination datacenters in the inter-datacenter network that guarantees the successful transmission by flow deadlines.

1.2.3 EAFR: An Energy-Efficient Adaptive File Replication System

A number of important challenges need to be overcome to achieve the aforementioned goals in file replication systems. First, the replication factor of each file should be dedicated assigned based on the request rate and availability of the file. Second, we need to maintain data availability when reducing energy consumption. Third, in order to avoid network bottlenecks, we need to effectively identify overloaded servers and dynamically change the transmission rate to prevent network congestion. In this dissertation, we propose an Energy-Efficient Adaptive File Replication System (EAFR), which incorporates three components. 1) It is adaptive to the time-varying file popularities to achieve a good tradeoff between data availability and efficiency. Higher popularity of a file on overloaded servers leads to more replicas and vice versa. 2) To achieve energy efficiency, servers are classified into hot servers and cold servers with different energy consumption, and hot/cold files are stored in hot/cold servers, respectively. 3) It selects servers with sufficient capacity (including network bandwidth and capacity) as replica destinations. 4) When replicating a file to a server, EAFR dynamically tunes the transmission rate to prevent potential incast congestion.

1.3 Contributions

We summarize our contributions of the dissertation below:

- We propose a lightweight system called CloudFog, which incorporates "fog" consisting of supernodes that are responsible for rendering game videos and streaming them to their nearby

players.

(1) We design a fog-assisted cloud gaming infrastructure. We leverage the hardware and bandwidth capacity of some idle machines from players and organizations, and deploy them as supernodes. These supernodes constitutes the "fog", which are responsible to stream game videos for nearby players.

(2) We propose a reputation based supernode selection strategy. In order to assign each player with a suitable supernode that can provide satisfactory game video streaming service, each player calculates reputation scores for all candidate supernodes according to previous interactions, and selects a supernode that has high reputation score, available capacity and low transmission delay.

(3) We propose a receiver-driven encoding rate adaptation strategy. In order to ensure the playback continuity even in network congestion, when a supernode streams a game video to a player, it adaptively changes the encoding rate of the video based on the segment size in the player's buffer according to the game's tolerance on delay and packet loss.

(4) We propose a social network based server assignment strategy. The communication between servers in a datacenter for generating game videos leads to latency. As social friends always play together [23], we assign social friends who usually play together to the same server, so their interaction will not trigger communication between different servers, thus reducing response latency.

(5) We propose a dynamic supernode provisioning strategy. When a large number of players join a game within a short time during peak hours, the cloud servers face a heavy burden. In order to deal with user churns and reduce server loads, we dynamically predict the number of players and then determine the number of pre-deployed supernodes based on the predicted value.

- We propose EcoFlow, an economical and deadline-driven video flow scheduling system to reduce the bandwidth payment costs of cloud providers.

  (1) To make the EcoFlow design more comprehensive, we use rate limiters to control that the flows are transmitted using the links' available bandwidth capacities.

  (2) At the beginning of the charging period, we set an initial charging volume on each link to reduce the scheduling latency.

(3) We provide discussion on how to deal with prediction errors and lack of prior knowledge in EcoFlow.

(4) We design both a centralized implementation and a distributed implementation of EcoFlow;

- We propose EAFR, an Energy-Efficient Adaptive File Replication System, which can reduce file read latency, power consumption and replication completion latency.

  (1) EAFR is adaptive to the time-varying file popularities to achieve a good tradeoff between data availability and efficiency. Higher popularity of a file on overloaded servers leads to more replicas and vice versa.

  (2) To achieve energy efficiency, we classify servers into hot servers and cold servers with different energy consumption, and hot/cold files are stored in hot/cold servers, respectively.

  (3) To reduce replication latency, we selects servers with sufficient capacity (including network bandwidth and capacity) as replica destinations.

  (4) We further propose three strategies to improve the performance of EAFR. First, when replicating a file to a server, EAFR dynamically tunes the transmission rate to prevent potential incast congestion. Second, when a compute node needs to read a file, EAFR uses a network-aware data node selection strategy to reduce file read latency. Third, when replica node failure occurs, EAFR uses a load-aware replica maintenance strategy to quickly create file replicas in other nodes.

Finally, we conducted extensive experiments on many testbeds, including PeerSim, PlanetLab, EC2 and a real-world cluster, which demonstrate the efficiency and effectiveness of our proposed methods. Experimental results show that compared to existing methods, CloudFog can increase user coverage, reduce response latency and bandwidth consumption; EcoFlow achieves the least bandwidth costs for cloud providers and transmits more video flows within their deadlines; EAFR is effective in reducing file read latency, replication time, and power consumption in large clusters.

1.4 Dissertation Organization

The rest of this proposed is structured as follows. Chapter 2 introduces the related works. Chapter 3 details the proposed method, extending cloud gaming for thin-client MMOG with high quality of experience. Chapter 4 presents EcoFlow, economical and deadline-driven video flow

scheduling system. Chapter 5 introduces EAFR, an energy-efficient adaptive file replication system. Finally, Chapter 6 concludes this dissertation with remarks on our future work.

Chapter 2

Related Work

Over the past few years, many works are proposed to solve the challenges in delivering and storing contents on the cloud. In this chapter, we present the related works on each of the proposed research problems in Chapter 1.

2.1 Cloud Gaming for Thin-Client MMOG

MMOG on the client-server architectures has gained much attention in the research communities in recent years. Common approaches of MMOG divide the virtual environment into regions and assign each region to different servers [21]. Bezerra *et al.* [20] proposed a kd-tree mechanism to partition the game environment into regions, and perform load balancing among multiple servers based on the distribution of avatars in the virtual world. Many works proposed to leverage the bandwidth contribution of peer-to-peer (P2P) networks to reduce server load [9]. Ahmad *et al.* [9] presents a P2P live video system to help players share screen-captured video of their games. Chen *et al.* [28] proposed a content-oriented pub/sub system that exploits the network condition and end-systems to enable efficient player management and decentralized information dissemination. These P2P and information dissemination methods cannot be directly applied to the context of cloud gaming, in which each player receives its own game video that cannot be shared with other players. Also, the players with thin clients may not be able to conduct rendering, computation and storage [36], which are offloaded to the cloud.

Previous works developed different cloud gaming systems. GamingAnywhere [56] is the first

open cloud gaming system with high extensibility, portability, and reconfigurability. Zhao *et al.* [117] designed a game cloud with a visualized cluster of CPU/GPU servers to reduce game latency of thin computers. Wang *et al.* [108] proposed to shift the burden of executing gaming engine from mobile devices to cloud servers, and developed a mobile gaming user experience model to characterize user experience. Hemmati *et al.* [52] presented a content adaptation encoding scheme, in which only the most important objects from the perspective of the player's activity are encoded in the scene and irrelevant or less important objects are omitted. EdgeCloud [35] augments the cloud infrastructure with a number of servers with specialized resources located near end-users to increase user coverage, these servers are responsible for computing new game state and rendering game video for players. The difference between EdgeCloud and CloudFog lies in the responsibility of newly added servers. In EdgeCloud, the addition of a small number of servers are used to store and compute game status and render new game videos; while in CloudFog, the storage and computation are carried out on the cloud, servers are only used to render new game videos and stream them back to the players. As the rendering work does not require high hardware configuration, given the same amount of revenue, *CloudFog* can deploy more servers than EdgeCloud by using proper incentives to motivate players or organizations to contribute their spare machines.

Early works also studied user experience in cloud gaming. Hobfeld *et al.* [54] discussed some technical challenges emerging from shifting gaming services to the cloud, and studied impacts caused by this change on user QoE. Jarschel *et al.* [63] conducted user studies to measure and model the QoE of OnLive during game play. Studies [63, 73] investigated how the response latency in cloud gaming affects QoE in various online games. There are also plenty of works on analyzing the challenges and benefits of system design in cloud gaming. Claypool *et al.* [37] studied detailed measurements of motion and scene complexity for a wide variety of video games, and measured the efficiency of streaming video games for thin clients. Choy *et al.* [36] demonstrated that the expansion of potential users for an on-demand gaming service is hindered by strict latency requirements, and indicated that the addition of a small number of servers can increase user coverage. Ojala *et al.* [82] studied a successful cloud gaming business model. They pointed out that deploying the games on cloud makes illegal copying practically impossible. Thus, cloud gaming can save the game developers from worrying about illegal copying.

In spite of the previous research efforts on cloud gaming, except deploying more datacenters which is costly, no other approaches have been proposed to handle its critical challenges. We propose

light-weight strategies to tackle the challenges to support thin-client MMOG.

2.2 Inter-Datacenter Video Flow Scheduling Systems

Recently, many methods have been proposed to schedule the inter-datacenter traffic in order to minimize the ISP bandwidth costs of cloud providers, which can be classified to two groups: store-and-forward and optimal routing path.

Store-and-forward. The methods in this group postpone the transmissions of the delay-tolerant data flow from peak hours to off-peak hours, so as to utilize the leftover traffic during off-peak hours. Laoutaris *et al.* [71, 72] proposed to employ a number of storage servers to collect delay-tolerant traffics and perform data transmission only when the destination datacenter is in predefined off-peak hours, so that the charging volume will not increase during peak hours. NetSticher [81] performs transmissions of delay-tolerant data between two datacenters only when both datacenters are in off-peak hours. If there are no common off-peak hours between both datacenters, it uses an intermediate datacenter that has an overlap in off-peak hours with the destination datacenter as a relay datacenter to store the delay-tolerant data temporarily.

Such store-and-forward transfer systems predefine off-peak hours of each datacenter. Delaying the transmission of delay-tolerant videos from peak hours to off-peak hours is a coarse-grained scheduling strategy. It does not fully utilize the link's available bandwidth capacity when actual traffic load is light during peak hours. Also, the transmission of a large number of non-delay-tolerant videos during the peak hours will increase the link's charging volume. Instead, EcoFlow is a fine-grained video flow scheduler which estimates the available bandwidth capacity on each link during a short time interval (i.e., 1 hour), and schedules the pending flows using a link's available bandwidth capacity in an earliest-deadline-first manner. The flows expected to missed their deadlines are rerouted to other under-utilized links so as not to increase the current charging volume.

Optimal routing path. The optimal routing path methods identify routing paths for video flows with the objective of minimizing the bandwidth payment costs. Multihoming is a scenario in which a user is connected to the internet through multiple links operated by different ISPs, and the links have different bandwidth capacities, availabilities and prices. In order to optimize the user's bandwidth costs and network performance in multihoming, Goldenberg *et al.* [48] used liner programming techniques to dynamically assign traffic among different links. Entact [116] applies a

route-injection mechanism to estimate the bandwidth costs of alternating paths that are not being currently used. Based on information of payment costs, traffic, and link capacity, it jointly computes the optimal routing path for online service providers. In order to reduce users' bandwidth costs in multihoming, Wang *et al.* [107] applied both dynamic programming algorithm and greedy algorithm to select links operated by different ISPs) to transfer the user data. Dhamdhere *et al.* [42] considered monetary cost and network availability in multihoming, and used the first fit decreasing algorithm to select an optimal set of ISPs. Jetway [45] aims to control the transmissions of video flows under the link's charging volume. When a video flow is expected to increase a link's current charging volume, the video will be split into subflows, and the subflows are transmitted on a multi-hop path to utilized the available bandwidth capacity of each link. These methods do not take advantage of the fact that some delay-tolerant videos are elastic to delay when scheduling the traffics. The charging volume of all option links will be increased when a large amount of videos need to be sent concurrently. Instead, EcoFlow takes advantage of different deadlines of flows and postpones the delivery of delay-tolerant videos to utilize a link's available bandwidth capacity when the current traffic load is high, so that the charing traffic volume will not further increase.

2.3 File Replication Systems in Data-Intensive Clusters

File replication is a common strategy to improve data reliability and availability in large clusters. HDFS [98], Lustre [102] and PVFS [83] maintain a constant number of replicas for each file, and replicas of the same file are placed in randomly selected servers. Many methods [5, 14, 109] have been proposed to improve the replication policy for different purposes. CDRM [109] adjusts the replication factor to maintain a required availability for each file under server failures based on the relationship between file availability and replication factor when servers have a certain probability to fail. Scarlett [14] aims to speed up the jobs by increasing the replication factor in the MapReduce systems. It is an off-line system that studies the file access patterns, and computes a replication factor for each file with a replication budget for load balance. In order to improve data locality in the MapReduce systems, DARE [5] replicates remote data into the local node when a map task processes data from remote nodes. It also applies a replication budge to limit the amount of replicas to save storage resource. Unlike the above replication works, EAFR aims to improve the data availability with the consideration of file popularity and file storage system efficiency.

Network bottleneck [115] is critical issues in data-intensive clusters but are neglected in previous file replication methods. Hedera [11] aims to maximize aggregate network utilization by collecting flow information from constituent switches. It studies the traffic demands and routing flows, and instructs switches to re-route traffic accordingly. Orchestra [34] studies the short-term traffic, then incorporates scheduling policies such as multipath routing and transfer priority at the transfer level to improve network performance inside clusters. These schedulers are based on the constraint that the traffic sources and destinations are already fixed, EAFR flexibly selects the servers with available network capacity to avoid network bottlenecks.

Energy-conservation in large-scale datacenters has drawn considerable research attention. Some studies [75, 114] aim to reduce the power costs by dynamically transitioning the servers to a sleeping state in datacenters. Recent research [13, 27, 74] proposes maintaining a minimal subset of nodes that are guaranteed to be on, and put other nodes to sleeping mode. This strategy ensures that a primary replica of each file is stored on active servers to provide service to file requests; however, it does not consider file popularity and replicas of popular file are also stored on inactive nodes. This generates a large number of replicas for popular files in order to ensure the file's immediate availability, and it suffers from degraded write-performance as the writes need to be executed on all servers storing the file replicas. GreenHDFS [65–67] trades performance for energy saving by logically separating the Hadoop cluster into hot and cold zones. Cold zone keeps low power consumption but provide less critical response for file accesses (i.e., long latency); while hot zone consumes more power and has strict performance requirements. It then uses data classification policies to place data onto a suitable temperature zone, that is, data that is frequently accessed by Hadoop framework is placed in hot zone, while unpopular data is place in cold zone. Different from these works, EAFR considers file popularity when allocating file replicas in order to save energy.

Chapter 3

CloudFog: Extend Cloud Gaming for Thin-Client MMOG with High Quality of Service

In this chapter, we explore how to extend cloud gaming for thin-client MMOG with high quality of experience. We first introduce our proposed CloudFog system, which incorporates multiple strategies to enhance its QoS. Experimental results from PeerSim and PlanetLab show the effectiveness and efficiency of CloudFog and our individual strategies in increasing user coverage, reducing response latency and bandwidth consumption.

3.1 Overview

Previous studies [36, 63] revealed that the uploading from the players to the cloud does not seriously affect the response latency, and downstream latency is an important factor for QoS [63], which is affected by the game video streaming rate. Thus, we aim to reduce the downstream latency by reducing the traffic transmitted from the cloud. In our design, game videos are streamed from nearby supernodes to players, instead of from remote game servers. As the computation of a virtual world for MMOG has a very high demand on server capacities [21], cloud is responsible for this task. Figure 3.1 shows our fog-assisted cloud gaming infrastructure. The fog is formed by supernodes,

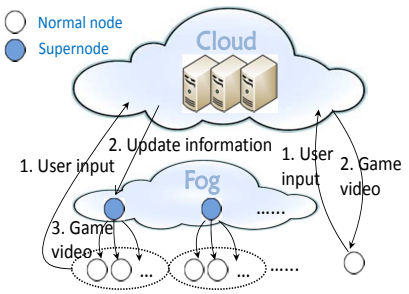


Figure 3.1: Fog-assisted cloud gaming infrastructure.

and normal nodes are connecting to their nearby supernodes. The normal nodes that cannot find nearby supernodes directly connect to the cloud.

We use n_i to denote a normal node, and sn_j to denote a supernode in the system. When each supernode is initially deployed, it is pre-installed with the *game client*. During the game playing, when node n_i makes an action (e.g., launching a strike or moving to a new place), this information is sent to the cloud server. The server collects action information from all involved players in the system and performs the computation of the new game state of the virtual world (including the new shape and position of objects and states of avatars). The cloud then sends the update information to the supernode of n_i (sn_j), which updates its virtual world accordingly. sn_j then renders game video for n_i based on $n_i's$ viewing position and angle. sn_j finally encodes the game video and stream it to n_i. As a player is close to its supernode in network distance, and the traffic from the cloud is significantly reduced, so the game video transmission delay is much shorter than that of downloading game video directly from the cloud as in the current cloud computing systems. Important notations used in this dissertation are listed in Table 5.1.

3.2 System Design of CloudFog

3.2.1 Requirements and Incentives for Supernodes

Rendering game video is relatively less hardware demanding than computation and communication in MMOG [35]; most modern computers with discrete graphics cards are sufficient to meet the rendering requirement. The nodes with sufficient hardware are chosen as supernodes, and the emerging technique of rendering multiple videos makes it possible for a supernode to suport multiple players simultaneously [1, 93]. As shown in [22, 79], desktop PCs in office are idle for about 12 hours per day, and 67% of desktop PCs remain powered on outside work hours (including nighttime). On

Table 3.1: Table of important notations.

n_i	a normal node
sn_j	a supernode
c_s	reward for one unit of bandwidth contributed by sn_j
$P_s(j)$	profit gained by supernode sn_j
c_j	supernode sn_j's upload capacity
u_j	sn_j's bandwidth utilization
$cost_j$	cost paid by sn_j's contributor in the same unit of c_s
$N(t)$	number of existing users at time t
n	number of normal nodes
m	number of supernodes
Λ	bandwidth usage for the cloud to send update information to one supernode
$G_s(j)$	game service provider's revenue gain by deploying sn_j
s_{ij}	overall reputation score of supernode sn_j evaluated by player n_i
r_k	k^{th} rating that n_i gives to sn_j
N_r	total number of ratings
λ	aging factor of ratings
d_k	age of rating r_k in days
q_i	game video quality for quality level i
b_{q_i}	game video bitrate for quality level i
$s(t_k)$	size of the video buffered at time t_k
r	number of video segments in the buffer
τ	game video segment size
β	game video bitrate adjust-up factor
θ	game video bitrate adjust-down threshold
ρ	latency tolerance degree
Γ	modularity of network communities

the other hand, the number of players in online games reaches a peak during the nighttime [25], which matches the idle time of office desktop PCs. So the supernodes can be contributed by different organizations that have idle computer resources, and game players that have powerful computers can also be selected as supernode candidates. Besides, game service providers can deploy their own supernodes by placing servers in different areas or rent on-demand resources from existing cloud providers like Amazon EC2. A game client of MMOG usually takes about 5-6GB storage space, and it is pre-installed in the supernode. The supernodes are required to be: 1) reliable, as malicious supernodes may distribute spam or virus that may degrade player experience or harm players' machines; 2) stable, supernodes need to provide stable support and notify the central server of game service providers before leaving the system; and 3) superior network connection, as supernodes need to stream game videos to players within short latency. To satisfy these requirements, organizations and individual players need to provide credentials to game service providers, game service providers will verify the information of supernode contributors and have contracts with them. The purpose of this contract is to ensure that supernodes can provide high QoS for players and will not leave the system abruptly during their service time. Contributing a machine as a supernode generates costs

of running the machine (e.g., electricity and maintenance costs). Therefore, to incentivize other organizations and players to contribute supernodes, an incentive mechanism is needed to reward supernodes based on the amount of upload bandwidth they contribute. The reward can be in the form of real money or virtual money for online games, and we use c_s to denote the reward for each bandwidth unit contributed by a supernode. An organization or a player considers to contribute a supernode only when it brings about certain profit, which is calculated by subtracting its running costs from its earned rewards. We use $P_s(j)$ to denote the profit gained by supernode sn_j:

$$P_s(j) = c_s \times c_j \times u_j - cost_j, \tag{3.1}$$

where c_j represents sn_j's upload capacity, u_j denotes sn_j's bandwidth utilization, and $cost_j$ denotes the cost paid by sn_j's contributor in the same unit of c_s. $P_s(j)$ quantifies the profit of contributing a supernode. Contributing a supernode is lucrative when $P_s(j)$ is greater than a certain threshold (different contributors set their own thresholds based on their expectations on profits). Then, the supernode's owner is motivated to contribute this supernode. Also, studies in [22, 79] found that 67% of desktop PCs remain powered on when they are idle, so there are powered idle PCs available to serve as supernodes. Companies and organizations are motivated to earn rewards by contributing these idle resources. Though some desktop PCs do not always serve players, as long as they function as supernodes in CloudFog, they can still receive a small amount of monthly sign up bonus. When they contribute bandwidth and support players, they can receive more credits. We will evaluate the effectiveness of this incentive mechanism in Section 3.3.

3.2.2 Economic Benefits for Game Service Providers

The game service provider needs to guarantee that the money spent on rewarding supernodes is smaller than the bandwidth costs saved by the contribution of supernodes. We use $N(t)$ to denote the number of existing users at time t. For simplicity, we omit t in the notations. Given the streaming rate of game video R, the total system demand for bandwidth equals $N \times R$. Suppose there are m supernodes, each having c_j upload capacity with utilization u_j. Then, supernode bandwidth contribution equals $B_s = \sum_{j=1}^{m} c_j \times u_j$. We use Λ to denote the bandwidth usage for the cloud to send update information to one supernode, and use n to denote the number of users that supernodes support. Then, in CloudFog, the bandwidth consumption for one player action for nodes connecting

to supernodes equals $\Lambda \times m$, and that for users directly connecting to the cloud equals $(N-n)R$. The bandwidth reduction (B_r^-) of CloudFog compared to current cloud computing system equals:

$$\begin{aligned} B_r^- &= N \times R - \Lambda \times m - (N-n)R \\ &= n \times R - \Lambda \times m \end{aligned} \tag{3.2}$$

Suppose c_c is the revenue gained by saving each server bandwidth unit, the goal of the game service provider is to maximize the saved cost by leveraging supernode bandwidth contribution, which can be formulated as below.

$$\begin{aligned} C_g &= \max(c_c \times B_r^- - c_s \times B_s) \\ &= \max\{c_c[n \times R - \Lambda \times m] - c_s \times B_s\} \end{aligned} \tag{3.3}$$

$$s.t. \quad \sum_{j=1}^{m} c_j \times u_j \geq n \times R \tag{3.4}$$

$$u_j \leq 1, \ \forall j \in \{1, 2, ...m\} \tag{3.5}$$

Equation (3.4) guarantees that the total supernode bandwidth contribution must reach the required node support bandwidth, while Equation (3.5) restricts the utilization of a supernode's upload bandwidth within its bandwidth capacity. In Equation (3.3), we see that given a specific number of n (*i.e.* the coverage of normal nodes is determined), saved cost C_g increases when m decreases; that is, a smaller number of supernodes lead to higher cost saving. For the game service provider, it should consider the pay and gain before deploying a supernode. Suppose a new supernode sn_j is deployed in an area; as a result, the coverage of players supported by supernodes is increased by ν new players. We use $G_s(j)$ to denote the game service provider's revenue gain by deploying sn_j, and $G_s(j)$ is estimated by:

$$G_s(j) = c_c[\nu \times R - \Lambda] - c_s \times c_j \times u_j. \tag{3.6}$$

If $G_s(j) > 0$, the cost of deploying supernode sn_j is surpassed by the benefit of bandwidth saved from the ν new players supported by sn_j.

3.2.3 Reputation Based Supernode Selection

3.2.3.1 Supernode Reputation Management

We define supernode sn_j's capacity as the maximum number of normal nodes that sn_j can support. Though supernodes are encouraged to contribute their computation and bandwidth resources to support other players' gaming activities, a supernode's quality of service to a given player is affective by three factors. First, there exits capacity heterogeneity among supernodes due to different upload bandwidth and computation power. A supernode may not be able to support all game video streaming requests with satisfactory QoS due to its limited available capacity. Second, different supernodes have different physical distances and transmission delays to a given player. Third, a supernode may not be willing to support players and deliberately throttles its upload bandwidth under certain circumstances. For example, when a supernode's owner runs many applications on the supernode, the owner may not be willing to support many players. Thus, jointly considering these three factors in selecting reliable supernodes is crucial to provide players with high QoS during gaming activities. To consider the third factor, we use a reputation system, which is an effective tool to guide the selection of supernodes that are willing to be cooperative in providing game video streaming service. In the reputation system, a player evaluates its supernode's reputation based on its performance in providing fluent game video streaming (i.e., playback continuity).

To facilitate the first two factors, the cloud stores the information of supernodes in the system in a table including their IP addresses and available capacities. When a newly joined node n_i requests a supernode, the cloud returns a number of supernodes that have available capacities and are physically close to player n_i by referring to the table. To do this, it first identifies the supernodes with available capacities. It then calculates the distance between each of the supernode candidates and the player, and selects a certain number of physically close supernodes. To calculate the distance, the cloud uses a supernode's IP address [92, 95] to determine its coordinate, and then uses the coordinate to calculate its distance from a player.

A physically close supernode may not guarantee a short transmission delay. Therefore, after the newly joined node n_i receives its close supernode candidates from the cloud, it tests the transmission delay to all of them. It then removes candidates with transmission delay greater than

its threshold L_{max_i}, which is determined based on the response latency requirement of the genre of its game [64]. This threshold is used to ensure that its supernode is capable of providing quick streaming support. From the remaining supernode candidates, node n_i then chooses the supernode with the highest reputation score. If there are no remaining supernode candidates, n_i directly connects to the cloud. Below, we describe the details of the reputation system.

To evaluate a supernode's willingness to be cooperative, a straightforward scheme is to evaluate a supernode's overall reputation by gathering opinions from all players interacted with this supernode [90]. However, this scheme is vulnerable to sybil attack [55], where a malicious supernode forges multiple identities and gains advantage by receiving high ratings from these identities. Also, it cannot prevent collusion in which a collective of nodes intentionally rate each other with high scores. To circumvent these problems, we let each player use its own evaluation without gathering opinions from other players.

Specifically, a player evaluates its supernode's performance in providing fluent game video streaming service after each game. It periodically calculates the overall reputation scores of its supernodes that provided it game video streaming service. As recent interactions between players and supernodes can more accurately reflect the supernodes' future performance than earlier interactions, we weight the ratings according to their ages when calculating a supernode's overall reputation score. Each rating is associated with an age measured by the number of days that have passed since the rating is given. We use s_{ij} to represent the overall reputation score of supernode sn_j evaluated by player n_i. It is computed as the weighted average of all ratings that sn_j receives from n_i:

$$s_{ij} = \sum_{k=1}^{N_r} r_k \lambda^{d_k}, \ (0 < \lambda < 1), \tag{3.7}$$

in which r_k is the k^{th} rating that n_i gives to sn_j, N_r is the total number of ratings, λ is the aging factor used to control the weights of ratings according to their ages, and d_k is the age of rating r_k in days. The reputation scores of the supernodes that have no previous interactions with the player equal to 0. The computation complexity of calculating reputation scores for all supernodes is $O(mnN_r)$, where m and n are numbers of supernodes and normal nodes.

3.2.3.2 Player and Supernode Churns Management

Recall that player n_i receives a number of supernode candidates from the cloud when it joins the system. In order to select supernodes with high reputations, it orders these candidates in descending order of their reputation scores. During the time when a player selects a supernode, a supernode candidate may be connected by more players and no longer has available capacity. To ensure that the selected supernode has available capacity, the player sequentially asks the supernodes in the ordered list whether it has available capacity. Once a supernode has available capacity, the player selects this supernode to connect to. If no supernode is selected after the player examines all supernode candidates, the player connects to the cloud for game video streaming.

Normal nodes probe their supernodes periodically for connection maintenance. When a normal node disconnects from its supernode, it first tries to find qualified supernode from its candidate supernode list by choosing the one with high preference ranking and available capacity. If it fails to find a new supernode from the candidate list, it contacts the cloud to find a new supernode using the method introduced above. When a new supernode is deployed, that is, a player or organization devotes a spare machine to earn rewards, the machine's location is identified based on the IP address. The cloud then notifies the normal nodes that are physically close to the new supernode, and these normal nodes will test the transmission delay to the newly deployed supernode. Finally, the suppernode will be added to the normal node's supernode candidate list if the transmission delay is less than L_{max_i}.

3.2.4 Receiver-driven Encoding Rate Adaptation

A player stores its received segments into its buffer while playing the game video. To guarantee the continuity of the video playback, the player needs to continuously fetch segments from the buffer and play. Game video bitrate affects the number of video segments received by a player during a unit time period, hence the player's playback continuity. Thus, we can adjust game video bitrate based on the size of buffered segments. The game video can be encoded to different bitrates based on the requirements on pixel size (resolution), hence the video quality level. A video segment with a higher quality level (i.e., a higher bitrate) leads to longer transmission latency. In order to illustrate the differences in video resolution, bitrate, latency requirement and tolerance degree for videos with different quality levels, we generate Table 3.2 as an example of parameter settings. Note that the parameter values in Table 3.2 are not precise, and the game service providers can

Table 3.2: Video parameters for different quality levels.

Quality level	Video resolution	Video bitrate	Latency requirement	Latency tolerance degree
5	1280x720	1800 kbps	110 ms	1
4	720x486	1200 kbps	90 ms	0.9
3	640x480	800 kbps	70 ms	0.8
2	384x216	500 kbps	50 ms	0.7
1	288x216	300 kbps	30 ms	0.6

set the parameters based on actual needs. As shown in Table 3.2, 500kbps corresponds to 384x216 resolution, and such a segment leads to 50ms latency. We use q_i ($i \in [1, ..., Q]$) to denote the video quality for quality level i and use b_{q_i} to denote the corresponding bitrate.

Different genres of games have different requirements on response latency [73]. Based on Table 3.2, if a game video has a latency requirement of 90ms, the supernode should use 1200kbps encoding bitrate, corresponding to a quality level of 4. To reduce the latency of the game video under unfavorable network condition, the supernode can choose encoding bitrates corresponding to quality level lower than 4; that is, sacrificing quality for lower latency. Due to unexpected network condition (e.g., network congestion), packets may be transmitted at a lower speed. Users may prefer fluent play of the game though the game video gets a bit blur when the encoding rate is reduced. To provide flexible options, users can also disable the encoding rate adaptation strategy before they start the game. In this case, the game video rate is fixed to the game's default video rate.

We aim to ensure that the playback rate is always lower than or equal to the segment downloading rate. When this condition cannot be satisfied, the video quality needs to be reduced by one level. When the size of buffered video at current quality level q_i is expected to reach the size of buffered video at quality level q_{i+1} (i.e., the downloading rate is faster than the playback rate), the current encoding bitrate b_{q_i} can be increased to $b_{q_{i+1}}$ to increase the video quality to q_{i+1}. Below, we explain the details of the adjustment operation. The estimated size of the video buffered at time t_k (denoted by $s(t_k)$) is calculated by:

$$s(t_k) = s(t_{k-1}) + (t_k - t_{k-1})(d(t_k) - b_p(t_k)), \tag{3.8}$$

where $d(t_k)$ and $b_p(t_k)$ denotes the downloading rate and video playback rate at time t_k. We use r to denote the number of segments in the buffer:

$$r = \frac{s(t_k)}{\tau} = \frac{s(t_{k-1}) + (t_k - t_{k-1})(d(t_k) - b_q(t_k))}{\tau}, \tag{3.9}$$

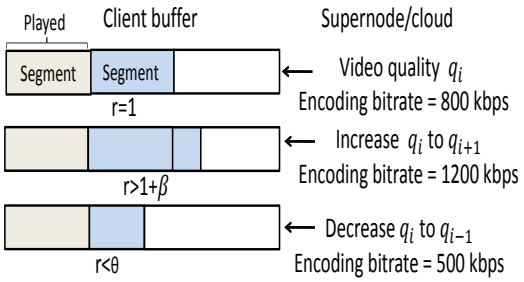


Figure 3.2: Receiver-driven encoding rate adaptation.

where τ denotes video segment size. If

$$r > 1 + \beta, \tag{3.10}$$

the video bitrate adjusts up. β is an adjust-up factor, and

$$\beta = max\{(b_{q_{i+1}} - b_{q_i})/b_{q_i}, \forall i \in [1, 2, ..., Q]\}. \tag{3.11}$$

β guarantees that the size of the buffered segments reaches that of the incremented quality level. When the video bitrate adjusts up, the user will not suffer from playback delay during the game. The adjust-down operation is performed if

$$r < \theta \ (\theta \leq 1), \tag{3.12}$$

where θ denotes adjust-down threshold. Formula (3.12) enables to proactively adjust down video bitrate to ensure the playback continuity in network congestion, in which the segment transmission time is typically much longer than usual. In order to prevent the fluctuation of the video bitrate for a client, the client can conduct the calculations of r for a number of times consecutively. The video bitrate is adjusted only when all results satisfy Formula (3.10) or Formula (3.12). Figure 3.2 shows an example of the encoding rate adaptation. When $r > 1 + \beta$ for several consecutive estimations, the supernode increases the video encoding quality by one level; from 800kbps to 1200kbps for the player. When $r < \theta$, the supernode decreases the video quality by one level; from 800kbps to 500kbps.

Different games have different latency-tolerant degree [73]. We consider this property to further enhancing the probability of meeting the response latency requirement for different games. Specifically, we require higher latency-sensitive games to have larger buffered video size for the encoding rate adjustment. We use $\rho \in [0, 1]$ to denote the latency tolerance degree; higher ρ means

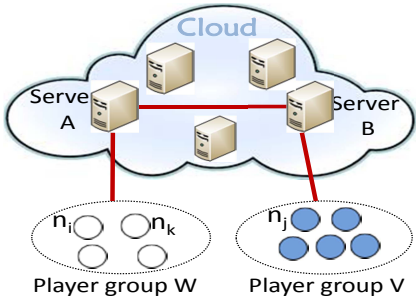


Figure 3.3: Server assignment based on social networks.

higher latency tolerance degree. We then change Formula (3.10) to $r > (1+\beta)/\rho$ and change Formula (3.12) to $r < \theta/\rho$ for triggering encoding bitrate adjustment. As a result, latency-sensitive (lower latency-tolerant) games have a higher r threshold while latency-tolerant games have a lower threshold for adjusting the encoding bitrate.

3.2.5 Social Network Based Server Assignment

A cloud datacenter consists of many servers, which cooperate to accomplish the computation and storage function of the datacenter. Therefore, when multiple players assigned to different servers interact with each other in the game (e.g., fighting each other in a battle), their servers need to communicate with each other in order to receive game states of all players and compute the game state of the virtual world. Online games involve intensive player interactions, if two players are assigned to different servers within datacenters, the interactions among these two players will lead to communications between the two servers. For example, in Figure 3.3, when player n_i is playing with player n_j, it will result in communication between server A and server B. Such server communications contribute to the response latency and degrade QoS. Thus, we propose the social network based server assignment strategy to reduce the interactions between servers by assigning players who are likely to play games together to the same server. In Figure 3.3, if n_i is playing with n_k, their interactions will not lead to server interactions.

When a new player signs up in CloudFog, if it builds friendships with other players, this new player is assigned to a server that most of its friends are allocated to; otherwise, it is randomly assigned to a server. To improve the accuracy of friend clustering and reduce the interactions between servers, we propose the social network based server assignment strategy that runs periodically (e.g., weekly) to reassign players to servers.

The players in the system can be represented by an undirected graph $G = (V, E)$; V is the

set of players and E is the set of edges between the players. $e_{ij} = 1$ if player n_i is a friend of player n_j. We use a set $F(i)$ to store all friends of n_i. The friendship relationship can be determined by two schemes: explicit friendship and implicit friendship. 1) Studies in [23] show that players tend to build friendship with each other in online computer games and players tend to play with their friends. By "friends" we mean that players who build friendship in the game. 2) CloudFog keeps record of each user's playing activities (e.g., who they are playing with, how long do they play), when the number of times that two players play together within the recent week CP_{ij} is larger than a threshold υ, we regard it as an implicit friendship. Thus, given z servers, this problem turns to finding z network communities (also sometimes referred to as modules or clusters), and the game information of each community is allocated in a distinguished server.

Modularity Γ is commonly used to evaluate the quality of resultant network communities [80]. It first defines a $z \times z$ symmetric matrix $\mathcal{Q}$, whose element q_{ab} is the fraction of edges connecting community A and B over $|E|$, then calculates Γ by:

$$\Gamma = \sum_{b=1}^{z}(q_{ab} - p_a^2) = tr(\mathcal{Q}) - ||\mathcal{Q}^2||. \tag{3.13}$$

$tr(\mathcal{Q})$ is the trace of matrix $\mathcal{Q}$; $p_a = \sum_{b=1}^{z} q_{ab}$; and $||X||$ is the sum of elements of matrix X. High value of Γ indicates a good community clustering, in which the game information of friends are likely to be allocated to the same server. Existing works generally partition a user and its friends to the same server by using replication [88], that is, a user's data is copied to multiple servers. These approaches are not applicable in the gaming area where a single copy of each player's data (e.g., profile data and game status data) is kept on a server to avoid the synchronization of user data on different servers. CloudFog first greedily assigns a player and its friends to the same community; then in order to optimize the community structure (i.e., increase the value of Γ), it repeatedly selects some players and switches their communities. The performance of community clustering and computation overhead can be controlled by setting different number of repetitions. We present the detailed steps below.

At first, all players are assigned to one community g_1, then we divide it into z communities using the following steps. 1) Randomly select a player n_i and put it and all its friends into a new community g_2. 2) Select a random play n_j from g_2, and put $F(j)$ (n_j's friends) into g_2. 3) Repeat step 2 until the number of player in g_2 is larger or equal to $|V|/z$. 4) Repeat step 2 and 3 to assign all players into z communities. 5) Calculate the modularity Γ_{pre} for current communities.

Randomly select player n_i and n_j from 2 random communities, swap the communities of $n_i + F(i)$ and $n_j + F(j)$, and calculate the current modularity Γ_{cur}. If $\Gamma_{cur} > \Gamma_{pre}$, swapping communities for players $n_i + F(i)$ and $n_j + F(j)$ leads to a better communities structure, so we keep the current structure; otherwise, we call it a **Miss** and rollback the swapping operation. 6) Repeat step 5 by h_1 times or until there are h_2 consecutive **Miss** ($h_2 < h_1$). Assuming $z^2 > |E|$, the computation complexity of calculating Γ_{cur} and Γ_{pre} is $O(z^2)$. Therefore, the computation complexity of this server assignment method is $O(h_1 z^2)$.

3.2.6 Dynamic Supernode Provisioning

In MMOGs, the number of online players generally varies with a diurnal pattern [84, 85]. When a large number of players join a game within a short time during peak hours, the surge in player arrival rate places a heavy burden on the cloud servers. MMOG designs should take into account the dynamicity of players and minimize the overhead of the cloud servers during peak hours. Provisioning supernodes to assist the cloud in game video streaming is an effective way to deal with player dynamicity.

In our dynamic supernode provisioning algorithm, the cloud pre-deploys a sufficient number of supernodes before the peak time to support players and removes these supernodes after the peak time. These supernodes serve newly-arrived players' requests and thus mitigate the peak bandwidth demand towards the cloud. A key challenge in our algorithm is to determine the number of supernodes that should be pre-deployed. If the game service provider reserves an excessive number of supernodes from players and organizations, some of them may be idle. If an insufficient number of supernodes are reserved, most requests for game video streaming will still be served by the cloud during peak hours.

We predict the number of players and then determine the number of pre-deployed supernodes based on the predicted value. Accurate prediction of online players is possible since previous works [84, 85] show that the workload of MMOGs has a regular weekly pattern and week-to-week load variations of players are less than 10%, for example, the trend of this Friday's online players mirrors that of last Friday. Thus, we can forecast the number of online players based on the data from previous weeks and reserve sufficient supernodes in advance. This forecasting and reservation process can be carried out at a frequency of every m-hour time window. Then, each week is divided into $24 * 7/m$ (denoted by T) time windows. We use $\hat{N}_t$ to denote the expected number of players in

time window t. $\hat{N}_t$ can be predicted from the number of players at the same time of last week, i.e., N_{t-T}. We use the seasonal ARIMA model [68], which is widely used to forecast time series with seasonal patterns, to forecast the number of players. It predicts $\hat{N}_t$ based on the data of the current time window (i.e., $t-1$) and data of the same time windows of last week (i.e., $t-T$ and $t-T-1$),

$$\hat{N}_t = N_{t-T} + N_{t-1} - N_{t-T-1} - \theta W_{t-1} - \theta W_{t-T} + \theta\Theta W_{t-T-1}, \tag{3.14}$$

in which θ is the moving average MA(1) coefficient and Θ is the seasonal moving average SMA(1) coefficient. $\{W_t\} \sim WN(0, \sigma^2)$ is a sequence of white noise with zero mean and variance σ^2. To support $\hat{N}_t$ number of players, the number of supernodes that need to deploy (denoted by Ns_t) is calculated by:

$$Ns_t = (1+\varepsilon)\hat{N}_t/\hat{C}, \tag{3.15}$$

where $\hat{C}$ is the average capacity of supernodes, and ε is the scale factor for the number of supernodes.

After determining the number of supernodes, the game service provider needs to select supernodes from available candidates. In order to maximize the number of players supported by the supernodes and utilization of supernodes' capacities, we need to select supernodes that are likely to receive a large number of service requests considering its location. Since the density of players in each area tends to be stable [118], a supernode that supports a large number of players previously has a high probability to attract a large number of players in the future. We leverage this intuition and select supernodes based on the number of players they support in the previous time slot. We use N_i to denote the number of players supported by supernode sn_i in the previous time slot. We then rank all supernode candidates by N_i in descending order. Finally, we create a supernode preference vector $V =< sn^1, sn^2, ..., sn^{N_s} >$. We select a supernode with rank j with probability P_j calculated by:

$$P_j = \frac{1/j}{\sum_{n=1}^{N_s} 1/n}, \tag{3.16}$$

where N_p is the number of supernodes. Finally, the pre-deployed supernodes have sufficient capacities to handle the request surge in their areas.

```java
package example.Game;

import java.io.BufferedWriter;

//GamingProtocal implements the peers' behavior in a cycle basic manner
public class GamingProtocol implements Cloneable, EDProtocol, CDProtocol {
    protected static String prefix = null;
    public GamingProtocol(String prefix) {    //initialize protocal parameters
        NODE_NUM = Configuration.getInt(prefix + "." + "NODE_NUM", 10000);
        ONLINE_TIME = Configuration.getLong(prefix + "." + "ONLINE_TIME", 3600*1000);
        SERVER_BW = Configuration.getInt(prefix + "." + "SERVER_BW", 10);
        ChannelProtocol.prefix = prefix;
    }
    //implement operations: get next packet from other peers or the cloud
    public Integer get_next_packet(Integer GameID, Integer PacketID, Node node) {
        Peer=SearchProvider(Integer GameID, Integer PacketID, Node node);
        if(Peer==-1){
            ContactCould(Integer GameID, Integer PacketID, Node node);
        }
        return Peer;
    }
    //define next cycle.
    public void nextCycle(Node node, int pid) {
        int count = 0;
        if (finished == 0) {
            for (Node n : GamingConnections) {
                GamingProtocol prot = (GamingProtocol) n.getProtocol(pid);
                if (prot.finished == 0) count++;
            }
            GameFinish[(int) node.getID()] += count;
        }
    }
}
```

Figure 3.4: Sample code of a PeerSim program.

3.2.7 Discussion on Security Issues of CloudFog

As supernodes can be contributed by players and organizations, attackers can control supernodes to reach their malicious goals such as gaining undeserved rewards and destroying normal functionality of the gaming system. For example, some supernodes may generate a large amount of junk files and send them to players so as to earn rewards from the game service provider; some supernodes can intercept or wiretap users' personal information; some supernodes may deliberately delay the transmission of game videos in order to destroy user satisfactions. These issues are critical but beyond the scope of this dissertation, and we will study them in our future work.

3.3 Performance Evaluation

3.3.1 Introduction to PeerSim Simulator and PlanetLab Real-world Testbed

We conducted experiments on the PeerSim [99] simulator and the PlanetLab [86] real-world testbed to evaluate the performance of CloudFog in comparison with other systems.

PeerSim is a single-threaded P2P simulator that can carry out large-scale experiments with millions of nodes. It simulates a typical P2P system where nodes join and leave continuously. In our experiments, we used a cycle-based simulation model that executes a simulation step in each cycle. Figure 3.4 shows a sample code of our developed PeerSim program. In this sample, we implement the *GamingProtocal* protocol whose parameters are initialized by the configuration file.

```java
package interCommunication;

import java.io.BufferedReader;

public class PeerCom  extends Thread {

    private ServerSocket bootStrap;
    ExecutorService execSvc;
    public PeerCom() {    //initialize thread pool and socket port
        try {
            execSvc = Executors.newFixedThreadPool(100);
            bootStrap = new ServerSocket(DatacenterTest.communicationPortPeer);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    public void run() {
        System.out.println("waiting for peers, my IP is "+GetLocalIP.getLocalHostIP().trim());
        while (true) {
            try {
                Socket socket = bootStrap.accept();  //wait for and accept sockets from peers.
                peerthread thread = new peerthread(socket);  //start peer operation once socket is established
                execSvc.execute(thread);     //add new thread to the execution pool.
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

Figure 3.5: Sample code of a Java program running on PlanetLab.

In this protocol, we can perform arbitrary actions by calling different methods (e.g., we implemented a method name *get_netx_packet* to let nodes fetch packets from other peers.) Finally, the status and useful information are passed to the next execution cycle by using the method *nextCycle*.

PlanetLab is an overlay testbed which contains nodes across the world and runs over the real internet. The nodes in PlanetLab are machines contributed by various academic institutions and universities. The advantage of PlanetLab lies in the fact that it can enable communications between two nodes from different areas, which is more realistic than simulation. A users of PlanetLab is allocated with a slice (i.e., computing and storage resources) and add nodes to its slice. A user then can run its programs on the distributed nodes. Figure 3.5 shows a sample code of our program running on PlanetLab. In this example, a node acts as a video game player. Its operations are defined by the method *run*. Particularly, it actively listens to sockets from other peers and establishes connections with them. Once a connection is established, it executes a thread to handle the interactions with the other peer.

3.3.2 Experimental Settings

We measured the performance in response latency, playback continuity and user coverage. *Basic CloudFog (CloudFog/B)* denotes the fog-assisted cloud gaming infrastructure without applying our proposed strategies; *Advanced CloudFog (CloudFog/A)* denotes our system with all proposed strategies. We compared *CloudFog* with the current cloud gaming model [56] (denoted by *Cloud*)

and *CDN* [35]. In *CDN*, a number of powerful CDN nodes are deployed to increase user coverage, which take over all the cloud's tasks (including storing and computing game status and rendering new game videos). The default number of main datacenters is 5 and 2 for all systems in simulation and PlanetLab, respectively. The number of servers within each datacenter is 5. As shown in Figure 3.18(b), the cost of renting a cloud server is twice as much as deploying a supernode, so the number of servers in *CDN* is set to 1/2 of the number of supernodes in *CloudFog*. We also conducted additional experiments for CDN with 45 and 8 randomly distributed servers in simulation and PlanetLab (denoted by *CDN-45* and *CDN-8*). Other default settings are: $\theta = 0.5$, $\lambda = 1$, $h_1 = 100$ and $h_2 = 10$. We used the statistics in [31, 57] for the distribution of download bandwidth in the simulation. To simulate real-world internet connections, a node's upload bandwidth capacity was set to 1/3 of its download bandwidth [3, 94]. In order to simulate a system with supernodes of various capacities, the capacity of supernodes (i.e., maximum number of normal nodes that can support) in the system follows a Pareto distribution [87, 110] with parameter $\alpha = 2$.

The experiment is divided into 28 cycles with each cycle representing one day's gaming activities; each cycle is further divided into 24 one-hour subcycles. According to [84, 85], we assume that 8pm-12am (i.e., subcycle 20 to 24) are peak hours when large number of online players are playing games. According to studies in [51], we randomly selected 50% nodes and 30% nodes to play for a period randomly selected from (0, 2] and (2, 5] hours a day, and let the remaining 20% nodes to play for a period randomly selected from (5, 24] hours a day. Inside one cycle, the game start time of each player is randomly selected from subcycle [1,19] with a probability of 30%, and randomly selected from subcycle [20,24] with a probability of 70%. To simulate supernodes' willingness in providing satisfactory streaming service, we randomly chose 1/5 and 1/10 supernodes that set their upload bandwidth at 80% and 50% of their capacities, respectively, with 50% probability in each cycle. After each experiment cycle, each player rates the supernode using the value of its game video playback continuity during this gaming activity. As defined in Figures 3.10(a) and 3.10(b), continuity is measured by the proportion of packets arrived within the required response latency over all packets in a game video. We use the first 21 cycles (i.e., 3 weeks divided into 126 time windows) as a warmup period to accumulate reputation scores for all supernodes. We record the number of online players for each subcycle and used this data to predict the number of online players. We then record the experimental results of the last 7 cycles and report the average value of these cycles.

Simulation settings. In the simulation, there were 10,000 game players (including online

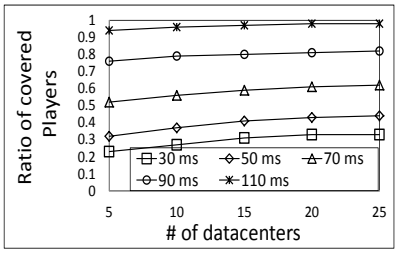
and offline players), 10% of which have the capacity to be supernodes. We randomly selected 600 supernodes for *CloudFog*. This is reasonable as the hardware requirement of servers in *CDN* is much more demanded than that of supernodes in *CloudFog*, thus, given the same amount of revenue, *CloudFog* can deploy more supernodes than the number of servers. The number of friends for each player follows power-law distribution with skew factor of 0.5 [4]. In order to simulate the dynamics of supernodes and players, the players join the system following the Poison distribution with an average rate of 5 players per second [111]. Each node leaves the system after it finishes playing and joins the system for the next experiment cycle. As in [17, 96, 101], the capacities of nodes follow Pareto distribution with a mean of 5 and shape parameter $\alpha = 1$.

We defined 5 games, their quality levels and latency requirements are shown in Table 3.2. When a player joins the system, if none of its friends is playing, it randomly chooses a game to play; otherwise, it chooses the game that has the largest number of its friends playing. OnLive provides gaming service at a frame rate of 30fps [62]. Thus, the frame rate of game videos in our experiment is set to 30fps. The communication latency between each pair of nodes was randomly selected from the ping latency traces from the League of Legends [2] based on each latency's occurrence frequency.

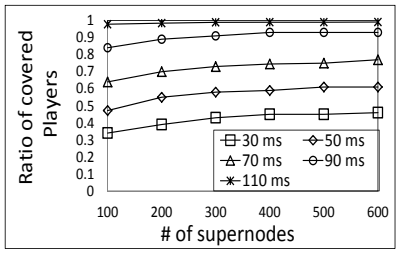
PlanetLab experiment settings. We used 750 distributed nodes nationwide, and 300 of them have the capacity to be supernodes. The nodes with IP 128.112.139.43 in Princeton University and IP 131.179.150.72 in the University of California, Los Angeles were set as cloud datacenters, due to their stable connection during the experiment. All other settings are the same as in the simulation.

3.3.3 Experimental Results for Overall Performance

We first tested the effectiveness of building datacenters in increasing user coverage in *CloudFog/B*. Recall that the general response latency requirement is 100ms [63]; 20ms is attributed to playout and processing delay and 80ms is the *network latency*. A user is covered by a datacenter or a supernode if the response latency is no more than the latency requirement of the user's game, and we measured the ratio of covered players as the number of players covered by a datacenter or a supernode over all players in the system. Figure 3.6(a) and Figure 3.7(a) show the ratio of covered players with different number of deployed datacenters and different network latency requirements of games on Peersim and PlanetLab, respectively. The figures illustrate that more datacenters lead to increased user coverage, as users are more likely to connect to close datacenters. Also, given a

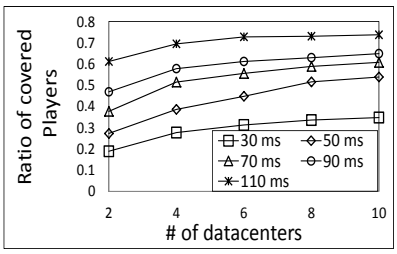


(a) User coverage VS # of datacenters.

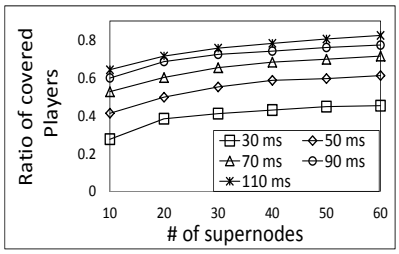


(b) User coverage VS # of super nodes.

Figure 3.6: Impact of # of datacenters and supernodes on PeerSim.



(a) User coverage VS # of datacenters.

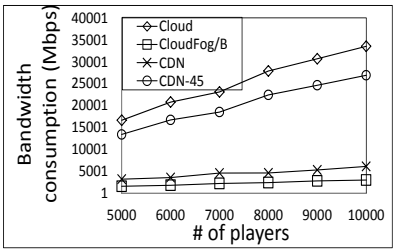


(b) User coverage VS # of supernodes.

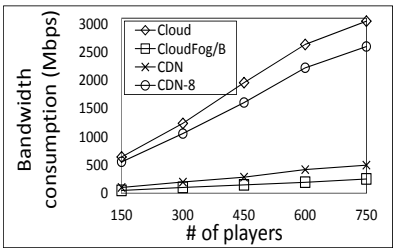
Figure 3.7: Impact of # of datacenters and supernodes on PlanetLab.

certain number of datacenters, stricter latency requirement leads to a smaller user coverage. In order to guarantee a better coverage of the user population, previous research suggested deploying more datacenters nationwide [35]. If OnLive chooses to build its own datacenters and building a medium size datacenter of approximately 300,000 gross square feet costs around 400 million dollars [18, 47], it would cost OnLive around 8 billion dollars to build 20 more datacenters; however, 25 datacenters can only cover 60% players with the general response latency requirement. Thus, increasing user coverage by deploying more datacenters is cost-prohibitive for game service providers. bandwidth costs represent In *CloudFog*, a game service provider can offer a small amount of monetary rewards as incentives to encourage supernodes, and user coverage can be increased by deploying supernodes.

We then examined the effectiveness of supernodes in increasing user coverage in *CloudFog/B* using 5 datacenters on PeerSim and 2 datacenters on PlanetLab. We see from Figure 3.6(a) that when the network latency requirement is 90ms, deploying 10 datacenters can increase about 10% user coverage than deploying 5 datacenters in PeerSim. Figure 3.7(a) reflects a similar trend as that in Figure 3.6(a), the two figures show that the effectiveness of increasing user coverage by deploying more datacenters weakens when the number of datacenters reaches a specific value. Figure 3.6(b) and Figure 3.7(b) show the ratio of covered players with different number of randomly selected supernodes and network latency requirements, Figure 3.6(b) shows that 100 supernodes can increase

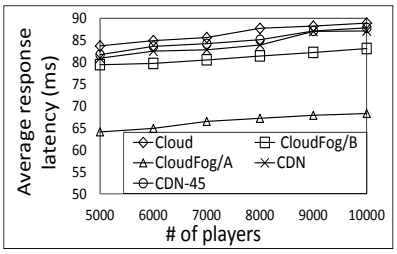


(a) The PeerSim simulator.

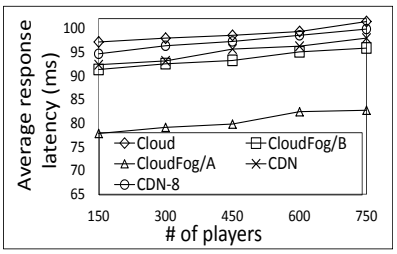


(b) The PlanetLab real-world testbed.

Figure 3.8: Server bandwidth consumption.



(a) The PeerSim simulator.



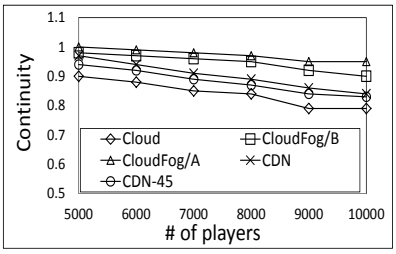
(b) The PlanetLab real-world testbed.

Figure 3.9: Response latency.

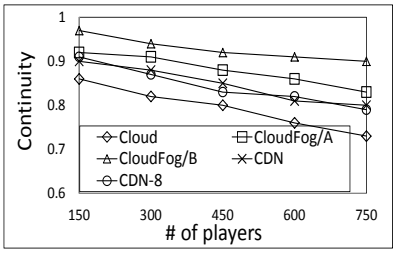
user coverage from 0.25 to 0.65 when the network latency requirement ranges from 110ms to 30ms. 200 supernodes can help achieve user coverage of deploying 25 datacenters. Figure 3.6(b) and Figure 3.7(b) show that instead of building datacenters, deploying supernodes is an effective alternative in increasing user coverage.

As players do not need to pay for bandwidth usage when they subscribe for internet services, we measure bandwidth consumption of different gaming systems from the side of cloud servers. Figures 3.8(a) and 3.8(b) show the bandwidth consumption of the cloud versus the number of players in the system. As *CloudFog/A* does not influent the bandwidth consumption of *CloudFog*, thus we use *CloudFog/B* to represent the bandwidth consumption of both *CloudFog/A* and *CloudFog/B*. We see that the result follows *Cloud*>*CDN*>*CDN-45*/*CDN-8*>*CloudFog/B*. The bandwidth consumption of *CDN* does not include those of additional servers. If we include them, *CDN*'s bandwidth consumption is similar to that of *Cloud*'s. *CDN* generates less bandwidth consumption than *CDN-45* and *CDN-8* as more servers are deployed to stream game videos to the players. *CloudFog/B* saves significant bandwidth consumption cost due to its employment of supernodes to stream game videos to the players. The cloud only needs to send update information rather than the entire game video to the supernodes.

Figures 3.9(a) and Figure 3.9(b) show the average response latency per player in different

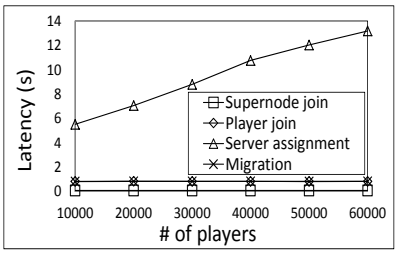


(a) The PeerSim simulator.

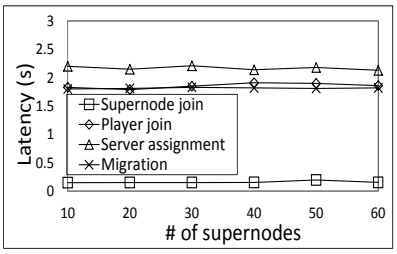


(b) The PlanetLab real-world testbed.

Figure 3.10: Playback continuity.



(a) The PeerSim simulator.

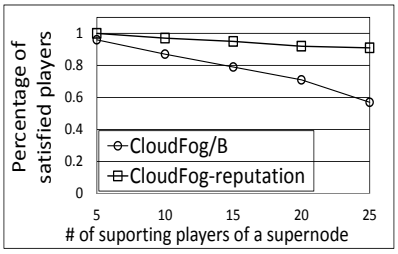


(b) The PlanetLab real-world testbed.

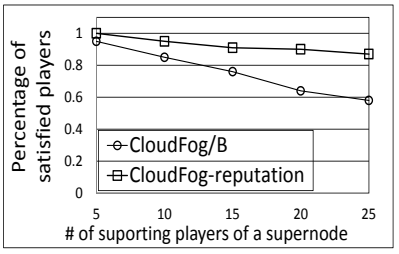
Figure 3.11: System setup latency and player join latency.

systems in PeerSim and PlanetLab, respectively. We see that *CDN-45* and *CDN-8* generate slight shorter response latency than *Cloud* due to the use of scattered servers, and users are more likely to connect to servers within a short distance. *CDN* further reduces the response latency as more servers are deployed. However, the improvement is not significant because the servers need to cooperate with each other to compute new game status, which lead to relatively long latency. *CloudFog/B* shows a slight reduction in response latency than that of *CDN*, which indicates the effectiveness of our fog-assisted infrastructure in reducing the latency. In *CloudFog*, users are supported by supernodes that are physically close to them. As the game video is streamed from supernodes to the users, instead of from servers that are physically far away. Thus, *CloudFog* is able to reduce the response latency for users. This result shows that our system not only reduces the response latency of the system of deploying many datacenters but also saves the prohibitive cost of building more datacenters. *CloudFog/A* further reduces the latency, which indicates the effectiveness of our proposed strategies in reducing response latency.

Video playback continuity is an important metric for QoS. We measured continuity by the proportion of packets arrived within the required response latency over all packets in a game video. Figures 3.10(a) and 3.10(b) show the average playback continuity of game videos when different number of players are playing games concurrently, which is a metric to measure weather a player

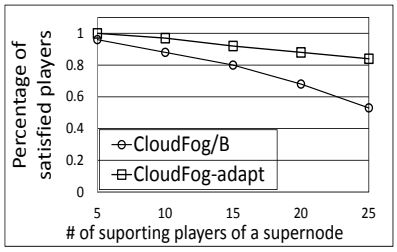


(a) The PeerSim simulator.

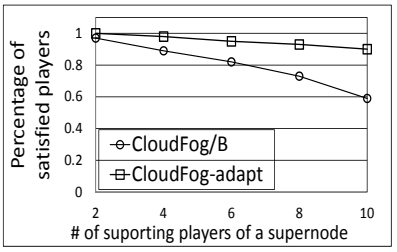


(b) The PlanetLab real-world testbed.

Figure 3.12: Effectiveness of reputation based supernode selection.



(a) The PeerSim simulator.

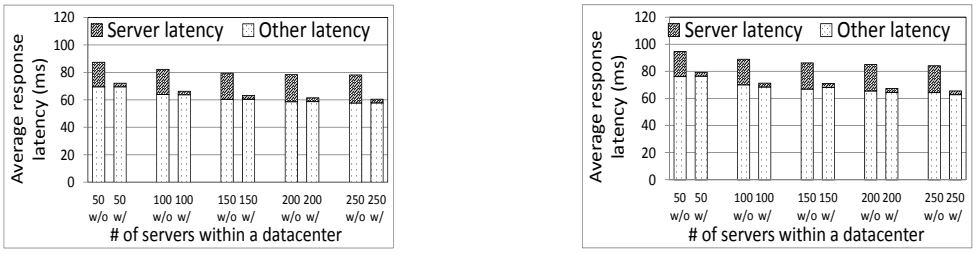


(b) The PlanetLab real-world testbed.

Figure 3.13: Effectiveness of encoding rate adaptation.

can enjoy smooth video playback. We see that *Cloud* yields the lowest playback continuity because there are only a small number of cloud servers, which may locate far away from some players. So most game videos need to be transmitted from remote servers to clients, thus large portion of packets cannot be received within the required response latency. *CDN-45* and *CDN-8* produce higher continuity than *Cloud* because players are supported by their nearby servers. *CDN* increases the playback continuity of *CDN-45* and *CDN-8* as players are more likely to find nearby servers when more servers are deployed. *CDN* generates smaller continuity than *CloudFog/B* and *CloudFog/A*, because not all users in *CDN* are able to connect to a nearby server due to the shortage of servers. So game video packets need to travel longer distance than that in *CloudFog*. *CloudFog/B* increases the continuity of *CDN* due to the effectiveness of the fog-assisted infrastructure, a large portion of users are supported by supernodes that are close to them. *CloudFog/A* provides an average of more than 90% continuity, with the contribution of all other proposed strategies.

We further tested: 1) server assignment latency, which is the time needed to allocate all players to cloud servers based on the social network based server assignment strategy; 2) average supernode join latency, which is average time from the time a supernode joins *CloudFog* until the time when it is connected to the cloud; 3) average player join latency, which is the average time from the time a player joins *CloudFog* until the time that it is connected to a supported supernode;



(a) The PeerSim simulator.

(b) The PlanetLab real-world testbed.

Figure 3.14: Effectiveness of social network based server assignment.

4) average migration latency, which is the average time needed for a player to connect to a new supernode when its supported supernode fails. When a player's supported supernode is out of service, the player needs to connect to a new supernode. We call the process of connecting to a new supernode a migration. In this experiment, we randomly chose 100 supernodes on PeerSim and 10 supernodes on PlanetLab, we then simulated supernode failures by disconnecting all players from these supernodes. In order to test the scalability of *CloudFog* on PeerSim, we varied the numbers of players from 10,000 to 60,000 and set the numbers of supernodes to 6/100 of players. Figure 3.11(a) shows the latency results on PeerSim. We see that when the numbers of players increase, the server assignment latency rises because the cloud needs to assign more players to servers, however, the server assignment latency does not increase rapidly. As the server assignment operation is conducted periodically (e.g., weekly), so the assignment latency does not compromise the QoS of *CloudFog*. The average supernode join latency remains low because supernodes only need to connect to the cloud; the the average player join latency remains constant since each player only needs to select a supernode from a small number of candidates. We also see that the migration latency is around 0.08 second, which is low. Because the game status is calculated on the cloud and supernodes do not need to store players' gaming information, there is no information transfer from the disconnected supernode to the new supernode. Thus, the migration overhead is small. During the migration, a player does not need to restart the game, and the game will resume after around 0.08 second. Figure 3.11(b) shows the latency performance when different numbers of supernodes are deployed on PlanetLab. We see that server assignment latency keeps stable as it is not affected by the number of supernodes. We also see that supernode and player join latency and migration latency stay low due to the same reason as in Figure 3.11(a). Figure 3.11(a) and Figure 3.11(b) indicate that the setup and dynamical reconfiguration of *CloudFog* can be completed within a short time.

3.3.4 Experimental Results for Proposed Strategies

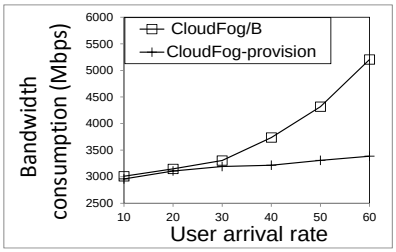
In the following, we show the effectiveness of each of our proposed strategies: i) reputation based supernode selection, ii) encoding rate adaptation, iii) social network based server assignment, and iv) dynamic supernode provisioning.

3.3.4.1 Performance of Reputation Based Supernode Selection Strategy

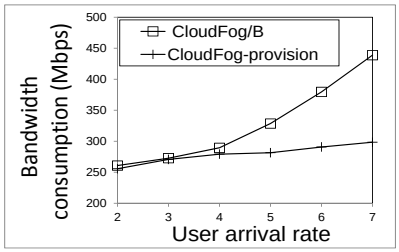
QoS is determined by packet loss rate and response delay. Thus, if a user can receive 95% of its game packets within the game's response latency, we consider this user as a satisfied player, and this definition is adopted in all figures within the dissertation. Figures 3.12(a) and 3.12(b) show the percentage of satisfied players with and without the reputation based supernode selection strategy, denoted by *CloudFog-reputation* and *CloudFog/B*, respectively. In *CloudFog/B*, among the final selected supernode candidates (introduced in Section 3.2.3), a player randomly selects a supernode from this set. We see that *CloudFog-reputation* significantly increases the percentage of satisfied players due to the reason that players are prone to select supernodes that can provide high QoS in streaming game videos. In *CloudFog-reputation*, each player evaluates supernodes' quality of service from previous interactions and selects the supernode that provides high QoS with high probability. Thus, the selected supernode is likely to support the player with high QoS in game video streaming. On the other hand, *CloudFog/B* randomly assigns supernodes to players. Though the assigned supernode is within the player's transmission delay threshold, the supernode may not be willing to provide all connected players with satisfactory streaming services.

3.3.4.2 Performance of Encoding Rate Adaptation Strategy

Figure 3.13(a) and Figure 3.13(b) show the percentage of satisfied players with and without (denoted by *CloudFog-adapt* and *CloudFog/B*) the encoding rate adaptation strategy, in PeerSim and PlanetLab, respectively. We see that *CloudFog-adapt* increases the percentage of satisfied users in *CloudFog/B*. The increase rate reaches 27% when the number of supported players of a supernode is 25 in the simulation. When the network condition is not good enough to support high quality streaming of game videos, this strategy decreases the video quality level to meet the response latency based on loss rate tolerance, thus increasing the number of satisfied players.

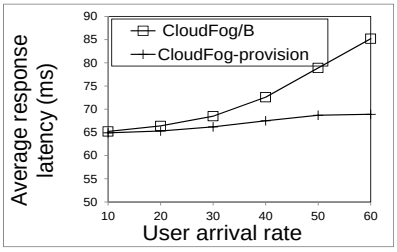


(a) The PeerSim simulator.

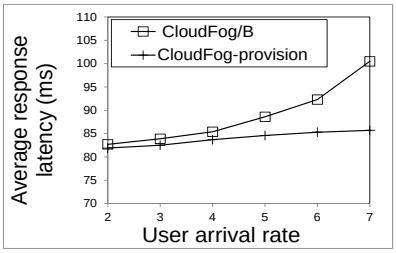


(b) The PlanetLab real-world testbed.

Figure 3.15: Effectiveness of the dynamic supernode provisioning strategy in reducing cloud bandwidth consumption.



(a) The PeerSim simulator.



(b) The PlanetLab real-world testbed.

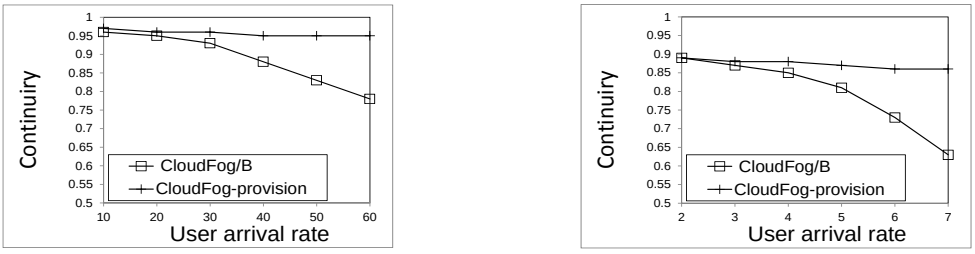
Figure 3.16: Effectiveness of the dynamic supernode provisioning strategy in reducing response delay.

3.3.4.3 Performance of Social Network Based Server Assignment Strategy

Figures 3.14(a) and 3.14(b) show the average response latency with (*w/*) and without (*w/o*) the social network based server assignment strategy on PeerSim and PlanetLab, respectively. In *w/o*, the users are randomly assigned to servers in a datacenter. We decompose the response latency to *server latency* (the communication latency among servers) and *other latency*. We see that *w/* produces about 20ms reduction in server latency, which leads to the reduction of overall response latency. This is because with this strategy, users that interact with each other in a game are more likely to be assigned to the same server within a datacenter, thus their interaction is less likely to involve communication among servers.

3.3.4.4 Performance of Dynamic Supernode Provisioning Strategy

In order to test the performance of *CloudFog* under user churns, we manually set different player arrival rates for peak hours and off-peak hours. In PeerSim simulation, we set the average player arrival rates during off-peak hours (subcycles 1-19) at 5 players/minute, and varied the average user arrival rate during peak hours (subcycles 20-24) from 10 to 60 players/minute with 10 players/minute increase in each step. In PlanetLab experiment, we set the average player arrival rates during off-peak hours at 1 players/minute, and varied the average user arrival rate during peak hours



(a) The PeerSim simulator.

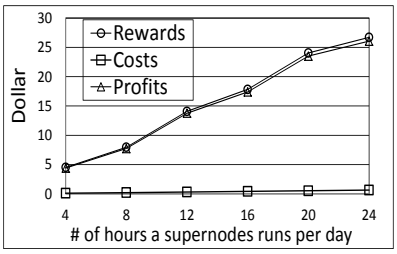
(b) The PlanetLab real-world testbed.

Figure 3.17: Effectiveness of the dynamic supernode provisioning strategy in increasing continuity.

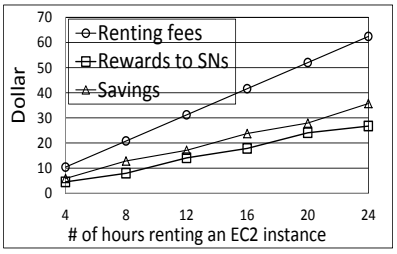
from 2 to 7 players/minute with 1 player/minute increase in each step. In *CloudFog/B*, the game service provider reserves a constant amount of supernodes, i.e., 400 supernodes in PeerSim simulation and 40 in PlanetLab experiments, while in *CloudFog-provision*, we dynamically set the number of supernodes according to the method in Section 3.2.6. The game service provider predicts the number of online players every 4 hours (subcycles) and reserves supernodes based on the prediction. Figures 3.15(a) and 3.15(b) show the cloud bandwidth consumption with and without the proposed dynamic supernode provisioning strategy. We see that as user arrival rate increases in *CloudFog/B*, cloud bandwidth consumption drastically rises in both PeerSim and PlanetLab experiments. This is due to the reason that *CloudFog/B* reserves a fixed number of supernodes regardless of the online player population. Then, when a large number of players are crowded into the system, most players cannot find support from supernodes and need to resort to the cloud for game video streaming. *CloudFog-provision* greatly reduces the cloud bandwidth consumption because it forecasts the potential rise in player population and reserve a sufficient number of supernodes in advance.

Figures 3.16(a) and 3.16(b) show the average response latency for *CloudFog/B* and *CloudFog-provision* in PeerSim and PlanetLab experiments, respectively. We see that *CloudFog-provision* can reduce the average response latency due to the reason that it reserves a sufficient number of supernodes in advance. When there are a large number of concurrent online players, these players can find support from supernodes that are physically close to them, so the response latency is reduced compared to downloading game videos from the cloud. While in *CloudFog/B*, a large portion of players rely on the cloud for game videos due to lack of supernodes, which generates long response latency as the cloud is physically far from the players.

Figures 3.17(a) and 3.17(b) show the average continuity for *CloudFog/B* and *CloudFog-provision* in PeerSim and PlanetLab experiments, respectively. We see that when user arrival rate increases, *CloudFog/B* leads to deteriorated average continuity for players. This is because when



(a) Rewards, costs and profits for supernodes.



(b) Renting fees and savings for a game service provider.

Figure 3.18: Economical incentives for supernodes and game service providers.

there are insufficient supernodes for an excessive number of concurrent players, the cloud needs to stream game videos to most players. As the video packets need to travel a long distance to the players, game interruption occurs when the packets cannot arrive within the game's required response latency. *CloudFog-provision* manages to sustain a high average continuity due to the same reason as in Figures 3.16(a) and 3.16(b).

These results verify that *CloudFog* is resilient to user churns. That is, when user arrival rate increases, the performance of *CloudFog* in providing high QoS in gaming activities will not be degraded.

3.3.5 Analysis of Incentives for Supernodes and Savings for Game Service Providers

As supernodes play important roles in *CloudFog*, we also evaluate the incentives for supernodes and the costs of deploying supernodes for game service providers. We select a random supernode and depict the profits earned by its owner. Assume that a supernode is a typical server that uses approximately 0.25kW electric power [46], and it is located in a region where the electricity cost is 10.8 cents/kWh, which is the US average price of electricity [7]. The hourly electricity cost of running the server is then $0.25 \times 0.108\ cents = 0.027$ dollar. We also assume that the game service provider pays 1 dollar for 1GB bandwidth a supernode contributes. Figure 3.18(a) shows the monetary rewards the supernode's owner earns from the game service provider, the costs of running the supernode and the owner's profits (calculate by Equation (3.1)) when the supernode runs for different number of hours. We see that the costs are trivial comparing to the rewards, so players and organizations are motivated to contribute their machines to earn profits.

For a game service provider, if it deploys 300 supernodes and all supernodes run 24 hours a

day for a full year, it needs to spend about 2.9 million dollars on rewarding the supernodes each year. While building a medium size datacenter costs around 400 million dollars, deploying supernodes rather than building extra datacenters is a more economical strategy for game service providers, and previous experimental results already show that supernodes are effective in providing high quality of service to users. Instead of building data centers, game service providers can rent instance resources from existing cloud providers. Assuming a game service provider rents a "g2.8xlarge" GPU instance from Amazon EC2 with 2.6 dollar per hour [38], we first plot the renting fees (denoted by *Renting fees*) in Figure 3.18(b). Compared to deploying a supernode with rewards (denoted by *Rewards to SNs*), we then plot the savings (denoted by *Savings*) for the game service provider by subtracting the *Rewards to SNs* from *Renting fees*. From Figure 3.18(b), we see that *CloudFog* is able to save game service providers' expenses.

Chapter 4

EcoFlow: Economical and Deadline-Driven Inter-Datacenter Video Flow Scheduling

In this chapter, we introduce our economical and deadline-driven video flow scheduling system (EcoFlow). We first introduce the objective of inter-datacenter video flow scheduling, which is to minimizes the bandwidth costs for cloud providers. We then provide an overview of EcoFlow and introduce our proposed EcoFlow in detail. Experimental results on PlanetLab and EC2 show that compared to existing methods, EcoFlow achieves the least bandwidth costs for cloud providers and transmits more video flows within their deadlines.

4.1 Overview

4.1.1 Objective of Inter-Datacenter Video Flow Scheduling

We consider a cloud with multiple geographically distributed datacenters operated by a single cloud provider. Every datacenter in the cloud is connected to all other datacenters. We use a complete directed graph $G = (V, E)$ to represent the inter-datacenter network, where V is the set of datacenters and E denotes the set of direct links connecting datacenters. For each link $e_{ij} \in E$,

Table 4.1: Table of important notations.

G	a graph of inter-datacenter network
V	a set of datacenters operated by a cloud provider
E	a set of links connecting each pair of datacenters
e_{ij}	a direct link connecting datacenter i and j
f_k	video flow k
f_k^I	an indirect video flow
f_k^D	a direct video flow
$F(t)$	a set of video flow at time t
$F^I(t)$	a set of indirect video flow at time t
$F^D(t)$	a set of direct video flow at time t
$F_{ij}(t)$	a set of video flow on link e_{ij} at time t
T_p	time window for traffic volume prediction
T_r	time window to record actual traffic volume
$[t_i, t_{i+T_p/T_r})$	time interval for traffic volume prediction
$[t_i, t_{i+1})$	time interval to record traffic volume, t_{i+1}-t_i=T_a
a_{ij}	bandwidth cost per unit traffic on e_{ij}
$v_{ij}(t_i, t_j)$	traffic volume on link e_{ij} at time interval $[t_i, t_j)$
$\hat{v}_{ij}(t_i)$	charging volume on link e_{ij} at time t_i
$P_{ij}^c(t_i)$	bandwidth cost on link e_{ij} at time t_i
$\tilde{v}_{ij}(t_i, t_j)$	estimated traffic volume on link e_{ij} at $[t_i, t_j)$
c_{ij}	maximum bandwidth capacity on link e_{ij}
$\Delta c_{ij}(t_i, t_j)$	available bandwidth capacity on link e_{ij} at $[t_i, t_j)$
t_k^{start}	starting time for video flow f_k
t_k^{end}	completion time for video flow f_k
P	reroute path for an indirect video flow
s_k	flow size of video flow f_k
d_k	transmission deadline of video flow f_k
$\hat{v}_{ij}^0(t_0)$	initial charging volume on each link e_{ij}
$\bar{V}(t_{end})$	average charging volume at time t_{end} of all links

we use a positive value a_{ij} to denote the cost per traffic unit from datacenter i to datacenter j, a non-negative value c_{ij} to denote the maximum link capacity, which is the maximum available transmission rate from datacenter i to datacenter j. We use $\hat{v}$ to denote the charging volume. Important notations used in this dissertation are listed in Table 5.1.

The objective of EcoFlow is to design a schedule strategy so that 1) the overall bandwidth cost is minimized, and 2) all flows can finish transmission before their deadlines. We use T_r to denote the time window to record traffic volume (i.e., 5-minute interval) for calculating the charging volume, and $P_{ij}^c(t_i)$ denotes the bandwidth cost at time t_i. $\hat{v}_{ij}(t_i)$ represents the charging volume on link e_{ij} at time t_i and $v_{ij}(t_{i-1}, t_i)$ denotes the total actual traffic during time interval $[t_{i-1}, t_i)$. Assume the charing period begins at time t_0, the bandwidth cost on link e_{ij} at time t_i can be calculated by:

$$P_{ij}^c(t_i) = \begin{cases} a_{ij}\frac{\hat{v}_{ij}(t_{i-1})}{T_r}(t_i - t_0) & \text{If } v_{ij}(t_{i-1}, t_i) < \hat{v}_{ij}(t_{i-1}) \\ a_{ij}\frac{\hat{v}_{ij}(t_i)}{T_r}(t_i - t_0) & \text{otherwise} \end{cases} \tag{4.1}$$

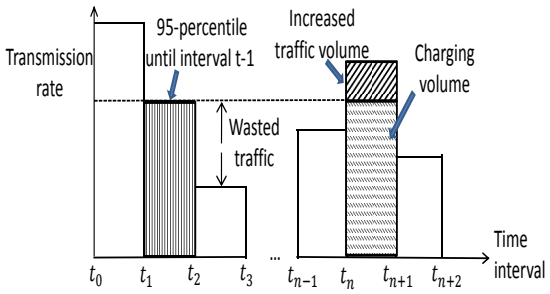


Figure 4.1: An example of bandwidth cost of inter-datacenter video traffic.

As shown in Equation (4.1), when the actual traffic volume during $[t_{i-1}, t_i)$ is smaller than the charging volume at time t_{i-1}, the bandwidth cost at time t_i is calculated by applying the cost function to the charging volume at time t_{i-1}; otherwise, the new bandwidth cost at time t_i is calculated by applying the cost function to the new charging volume at time t_i. The goal of the cloud providers is to minimize the bandwidth costs on all inter-datacenter links: $\min \sum_{\forall e_{ij} \in E} P_{ij}^c(t)$.

4.1.2 Strategy of EcoFlow

Constrain charging volume. Many recent methods try to constrain the charging volume on each link to reduce the bandwidth costs. We first present an example to explain the basic idea of these methods. Figure 4.1 shows an example of an inter-datacenter link's bandwidth cost under the 95th percentile charging model. The traffic volume in time interval $[t_1, t_2)$ is the 95th percentile value until time t_n, and is marked as the charging volume that needs to be paid by the cloud provider at t_n. When a larger traffic volume v_{ij} comes up in time interval $[t_n, t_{n+1})$, it becomes the new charging volume at time t_{n+1}. Then, from time t_0 to t_{n+1}, the unused bandwidth below v_{ij} is wasted, that is, the cloud provider does not fully utilized the charging volume. Given this observation, a feasible way to reduce bandwidth cost is to maximize the utilization of the charging volume at different time intervals. For this purpose, when the bandwidth needed is greater than current charging volume, EcoFlow postpones the delivery of later-deadline videos to the time when the traffic load is light. For example, the increased traffic volume in time interval $[t_n, t_{n+1})$ can be postponed to time interval $[t_{n+1}, t_{n+2})$. In this way, when a fixed amount of video flow is transmitted between two datacenters over a period, the 95th percentile of video volumes over all time intervals is minimized.

Three steps of EcoFlow. Specifically, EcoFlow schedules the video flow transfers on a link to different time slots or to other links in order to fully utilize the charging volume while

guaranteeing the successful flow transfer within deadlines. The EcoFlow scheduling mechanism can be divided into three steps. We first briefly introduce the three steps using an example in Figure 4.2, which demonstrate the flow scheduling on two links.

Step 1: **available bandwidth capacity estimation.** We use T_p to denote the time window used to estimate the available bandwidth capacity on each link, and use T_r (T_r<T_p) to denote the time window to record traffic volume in current charging model. Based on historical data, we estimate the total volume of video traffic needed to be transmitted on each link during time interval $[t_0, t_n)$, $t_n - t_0 = T_p$, denoted by $\tilde{v}(t_0, t_n)$. Assume link e_1's charging volume at time t_0 is $\hat{v}_1(t_0)$, it then can transfer a volume of $\hat{v}_1(t_0) \times T_p/T_r$ video during time interval $[t_0, t_n)$. We define **a link's available bandwidth capacity** as the maximum transmission rate that can be used to transfer videos without increasing the current charging volume during a certain time interval. We then calculate the available bandwidth capacity $\Delta c_1(t_0, t_n)$ on link e_1 during time interval $[t_0, t_n)$:

$$\Delta c_1(t_0, t_n) = \hat{v}_1(t_0)/T_r - \tilde{v}_1(t_0, t_n)/T_p. \tag{4.2}$$

Step 2: **deadline-driven flow scheduling.** On each link, the pending video flows are scheduled on an earliest-deadline-first base. When the traffic capacity is fully occupied at the current interval, we postpone the transfer of flows with later deadlines to later time interval but still guarantee their deliveries by deadlines. On link e_2, the transmission of flow f_{21} fully utilizes the available bandwidth capacity on link e_2 in time interval $[t_0, t_1)$, so f_{22} with a later deadline than f_{21} will be sent after f_{21} finishes transmission. However, when f_{24} is scheduled after f_{23}, its expected transmission time is at t_5, which is later than its deadline. We divide f_{24} into two subflows: f_{24}^D and f_{24}^I. On link e_1, all pending videos are scheduled to finish transmission before t_3, its available capacity during $[t_3, t_n)$ is not utilized (highlighted in dashed fill). We call the available bandwidth capacity that are not utilized during $[t_3, t_n)$ **extra bandwidth capacity ($\delta c_1(t_3, t_n)$)**, $\delta c_1(t_3, t_n)$=$\Delta c_1(t_0, t_n)$. We define the links with extra bandwidth capacity during a time interval as the under-utilized links. The extra bandwidth capacity on link e_1 can be utilized to reroute subflow f_{24}^I from e_2 by its deadline.

Step 3: **routing path identification.** For the video rerouting, we aim to identify an alternating path that has extra bandwidth capacity to transmit the video by its deadline. To this end, we reply on the Dijkstra's algorithm [43] and propose a path identification method.

Advantages of EcoFlow. We then use an example to show the advantage of EcoFlow

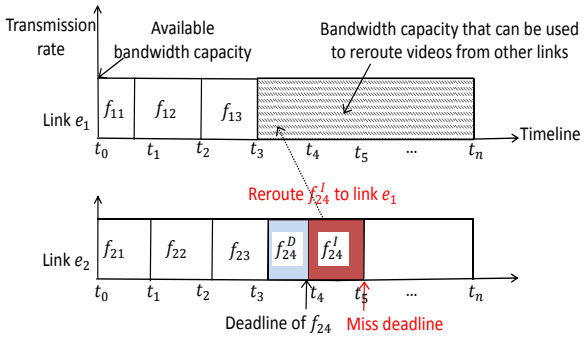


Figure 4.2: An overview of EcoFlow.

compared to existing methods. In Figure 4.3, the time interval in X axis is one second, that is, t_1-t_0=1s. Y axis denotes the available bandwidth capacity on a link. Four videos of 1GB in size need to be transferred on a link, and the link's available bandwidth capacity is 4Gb/s. We assume that time interval $[t_0, t_3)$ is the link's peak hours, while $[t_3, t_8)$ is in the link's off-peak hours defined in the store-and-forward methods [71,72,81]. The transmission requests of video flows f_1, f_2, f_3 and f_4 arrive at the source datacenter at time t_0, t_1, t_2 and t_3. All four videos are delay-tolerant with transmission deadlines at time t_6, t_7, t_8, and t_9, respectively. In the store-and-forward method, the delay-tolerant videos temporarily wait during peak hours during time interval $[t_0, t_3)$. As no video transmission is performed during peak hours, the link's bandwidth capacity is wasted. The videos are sent out during off-peak hours in $[t_3, t_8)$. However, as a total amount of 2.5 GB data can be transmitted using the link's available bandwidth capacity during the off-peak time, thus only 2.5 videos can finish transmission.

We take Jetway [45] as a representative method of the routing path optimization methods. In JetWay, all videos are sent out when their transmission requests arrive at the source datacenter. The transmission rate of each video flow is calculated by dividing the size of video by the time span between the transmission request arrival time and the video's deadline. Thus, each video is transmitted at the rate of 1GB/6s=1.33Gb/s. f_1, f_2 and f_3 will be transmitted at the rate of 1.33Gb/s at time t_0, t_1 and t_2, respectively. When f_4 arrives at time t_3, the link's bandwidth capacity is fully occupied by other videos, then f_4 has to be rerouted to other links. In this example, the bandwidth capacity marked in blue is wasted.

In EcoFlow, all videos are transmitted using the available bandwidth capacity, and the transmission of later-deadline videos will be postponed to future time slots if current traffic increases the charging volume. In this example, f_2 will be postponed until f_1 finishes its transmission, and

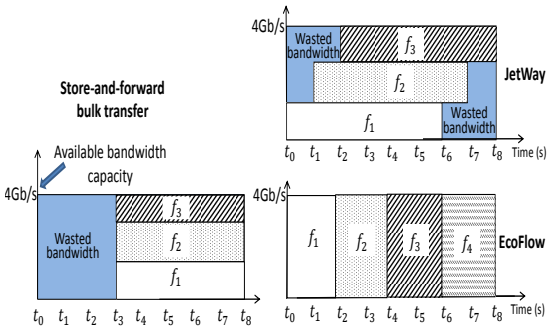


Figure 4.3: A comparison of bandwidth utilization between different methods.

f_3 and f_4 are then transferred one by one. All four videos can be transmitted before their deadlines and the link's bandwidth capacity is fully utilized. Therefore, EcoFlow can fully utilize the link bandwidth capacity to reduce the bandwidth payment cost of existing methods.

4.2 System Design of EcoFlow

4.2.1 Available Bandwidth Capacity Estimation

When the total traffic volume at current time interval $[t_i, t_{i+1})$ $(t_{i+1} - t_i = T_r)$ exceeds the current charging volume, EcoFlow postpones the transmission of later-deadline videos to utilize the link's available bandwidth capacity in future time intervals. In order to predict whether a link has enough bandwidth capacity to transmit these videos, EcoFlow estimates each link's available bandwidth capacity during T_p by two steps: 1) traffic volume prediction during T_p, and 2) available bandwidth capacity estimation, which is the maximum volume of traffic a link can transfer without further increasing the current bandwidth cost.

Traffic volume prediction Exponentially weighted moving average (EWMA) [77] is widely used for prediction for a given series of data points. Note that the traffic transmitted on link e_{ij} includes traffics transmitted from datacenter i to j and from datacenter j to i. We use EWMA to estimate the traffic volume during $[t_i, t_i + T_p]$ on link e_{ij} (denoted by $\tilde{v}_{ij}(t_i, t_i + T_p)$) based on the actual historical traffic volume (denoted by $v_{ij}(\cdot)$):

$$\tilde{v}_{ij}(t_i, t_i + T_p) = \beta \times v_{ij}(t_i - T_p, t_i) + (1 - \beta) \times \tilde{v}_{ij}(t_i - T_p, t_i). \tag{4.3}$$

$\tilde{v}_{ij}(t_i - T_p, t_i)$ is the estimated traffic volume in time interval $[t_i - T_p, t_i)$, and β $(0 < \beta < 1)$ is a constant used to control the degree of weighting decrease.

Available bandwidth capacity estimation From historical flow records, we calculate the charging volume on link e_{ij} at time t_i, $\hat{v}_{ij}(t_i)$. Thus, during time interval $[t_i, t_i + T_p)$, a total volume of $\hat{v}_{ij}(t_i) \times T_p/T_r$ video traffic can be transferred under the current bandwidth cost. Given estimated traffic volume $\tilde{v}_{ij}(t_i, t_i + T_p)$, we can calculate the available bandwidth capacity in time interval $[t_i, t_i + T_p)$:

$$\Delta c_{ij}(t_i, t_i + T_p) = \min\{c_{ij}, \hat{v}_{ij}(t_i)/T_r - \tilde{v}_{ij}(t_i, t_i + T_p)/T_p\}. \tag{4.4}$$

When $\Delta c_{ij}(t_i, t_i + T_p) > 0$, the current charging volume on link e_{ij} is larger than the expected traffic volume, and the available bandwidth capacity can be used to reroute video flows from other links.

4.2.2 Deadline-driven Flow Scheduling

Like existing works [45, 112] that assume the existence of a centralized server connecting to all datacenters that functions as a scheduler to schedule the video flows in all datacenters, we first introduce EcoFlow in a centralized manner. We will further introduce a distributed way to realize EcoFlow in Section 4.2.8.

In order to maximize the number of videos that can be transmitted by their deadlines, we use the earliest-deadline-first strategy, that is, video with the earliest deadline will be put at the front of the sending queue. The network scheduler maintains a sending queue $Q(t_i)$=($< f_1, d_1, s_1 >$,$< f_2, d_2, s_2 >$,... $< f_m, d_m, s_m >$) to store all pending flows on each link e_{ij} at time t_i, which are ordered based on their deadlines. Note that flows on link e_{ij} includes all flows that are transmitted bidirectionally between datacenter i to j (i.e., from i to j or from j to i). Each triple $< f_k, d_k, s_k >$ in $Q(t_i)$ contains the flow information of f_k, where d_k and s_k are the deadline and size of f_k, respectively.

All pending videos in $Q(t_i)$ are sent out sequentially, that is, f_k will be sent only when all videos f_1, f_2, ..., f_{k-1} have finished transmission. As we see from Figure 4.3, when a number of videos are transmitted on a link simultaneously and their cumulated transmission rate is less than the link's available bandwidth capacity, the bandwidth resource of this link is wasted as the extra bandwidth capacity is not utilized. Thus, EcoFlow aims to maximize the bandwidth utilization by sending video at a rate that fully utilizes the available bandwidth capacity. The estimated flow

transmission time for flow f_k can be computed as:

$$T_k = \mathcal{A}/\Delta c_{ij}(t_i, t_i + T_p). \tag{4.5}$$

$\mathcal{A} = \sum_{f_p \in F_{<k}} s_p$, where $F_{<k}$ is a subset of flows in $Q(t_i)$ that have earlier deadlines than flow f_k (including f_k), and $\Delta c_{ij}(t_i, t_i + T_p)$ is the estimated available bandwidth capacity on link e_{ij} in time interval $[t_i, t_i + T_p)$. As the flows in $Q(t_i)$ are sent sequentially, the flow completion time for f_k is t_k^{end}:

$$t_k^{end} = \begin{cases} t_i + T_k & \text{If k=1} \\ t_{k-1}^{end} + T_k & \text{otherwise} \end{cases} \tag{4.6}$$

The flow start time for f_1 is t_i. For flow f_k ($k > 1$), the flow start time is the completion time of the previous flow, that is, $t_k^{start} = t_{k-1}^{end}$. d_k is the deadline of video f_k. When $t_k^{end} \leq d_k$, flow f_k is expected to finish transmission before its deadline, and we call it a **Direct Flow** (DF). When $t_k^{end} > d_k$, flow f_k is likely to miss the transmission deadline under the expected bandwidth capacity. Then, f_k can be split to two subflows: f_k^D and f_k^I. f_k^D is the volume with size s_k^D that can be transmitted directly on link e_{ij} before its deadline; while f_k^I is the residual volume with size s_k^I (s_k^I=$s_k - s_k^D$), which should be rerouted in an alternating path before its deadline. We call f_k^I an **Indirect Flow** (IF).

$$s_k^D = max(0, d_k \times \Delta c_{ij}(t_i, t_i + T_p) - \mathcal{A}). \tag{4.7}$$

Using the alternating routing path identification method in Section 4.2.7, we identify alternating paths for each indirect flow f_k^I which can transmit f_k^I before its deadline.

After all IFs find alternative paths, the scheduler creates a schedule table for all pending flows, in which each item is expressed in a quadruple $S(t_i) =< f_k, S, D, t_k^{start} >$, which denotes the flow ID, source datacenter, destination datacenter and flow start time. This table is used to guide the transfers of flows initiated from all datacenters.

4.2.3 Alternating Routing Path Identification

$F^I(t_i)$ denotes the set of all IFs in the network at time t_i, which are sorted by their deadlines in ascending order. Assume f_k^I is an IF from datacenter i to datacenter j, in this section, we describe how the scheduler identifies an alternating routing path P for f_k^I, $P = (v_1, v_2, \ldots, v_p)$. The

centralized scheduler identifies an alternating path for each IF in an earliest-deadline-first manner.

The selected alternating path P uses extra bandwidth capacities of its constituent links to transmit f_k^I, with the requirement of finishing the transmission before its deadline. The path that can transmit f_k^I with the minimum transmission time among all possible alternating paths is the best path to satisfy this requirement. Thus, we first identify the alternating path P that leads to minimum transmission time. If the identified path can transmit f_k^I before its deadline, we reroute f_k^I to this alternating path; otherwise we split f_k^I into two parts: f_{k1} and f_{k2}, where f_{k1} is the part of f_k^I that are expected to finish transmission on P. We reroute f_{k1} along P, and identify another alternating path for f_{k2} by using the same process. If the new identified alternating path cannot transmit f_{k2} before its deadline, f_{k2} is further split into two parts: f_{k2}^1 and f_{k2}^2. f_{k2}^1 is transmitted on the new alternating path and f_{k2}^2 is transmitted on e_{ij} by increasing the charging volume on e_{ij}.

When f_k^I is transmitted on path P, its transmission rate is the minimum extra bandwidth capacity on all P's constituent edges [44], that is $\min_{\forall i \in (1,p-1)}\{\delta c_{i,i+1}(t_i, t_i+T_p)\}$. f_k^I's transmission completion time t_k^{end} is calculated by:

$$t_k^{end} = s_k^I / \min_{\forall i \in (1,p-1)}\{\delta c_{i,i+1}(t_i, t_i + T_p)\} + t_i. \tag{4.8}$$

s_k^I denotes the flow size of f_k^I and $\delta c_{i,i+1}(t_i, t_i + T_p)$ denotes the extra bandwidth capacity on link $e_{i,i+1}$. We then express the requirement that the transmission of flow f_k^I on path P would be finished before its deadline by: $t_k^{end} \leq d_k$. The path that can transfer f_k^I with the minimum transmission time is the best path to satisfy this requirement, which is shown in Equation (4.9).

$$\min_{\forall P}\{t_k^{end}(P)\} \tag{4.9}$$

As Dijkstra's algorithm can be adopted to find the path that can transmit f_k^I with the minimum transmission time, we develop a modified Dijkstra's algorithm shown in Algorithm 1 to identify an alternating routing path for f_k^I. In this algorithm, we input the flow information including its size, deadline, source datacenter and destination datacenter, together with network information including all link's extra bandwidth capacity. Algorithm 1 will return an alternating path P for f_k^I, and splits f_k^I into two parts if P cannot finish transmission before its deadline. In algorithm 1, we modify the original Dijkstra's algorithm by adding line 25 to 29, which is to check whether the identified alternating path P meets the deadline requirement. If the identified path can transmit f_k^I

before its deadline, alternating path P is returned (Line 25); otherwise f_k^I is split into f_{k1} and f_{k2} (Line 26-29). The simplest implementation of the Dijkstra's algorithm requires a running time of $O(|E| + |V|^2) = O(|V|^2)$.

Algorithm 1: Pseudocode for identifying alternating path for flow f_k^I.

```
1: Input: G = (V, E); s_k, δc_ij(t_i, t_i + T_p), ∀ij ∈ E;
2: Output: alternating path P from source i to destination j
3:    for each vertex u in V
4:       rate[u] := 0   //The maximum transmission rate on each path
5:       pre[u] := null   //Record last hop on the path
6:       t_k^end[u] := t_i   //Transmission completion time
7:       Q+ = j   //Q is a temporal set
8:    end for
9:    rate[i] := infinity
10:   while Q is not empty do
11:      u := vertex in Q with max rate[u]
12:      remove u from Q
13:      for each neighbor v of u with δc_uv(t_i, t_i + T_p) > 0 do
14:         t_k^end[v]:=s_k^I/min{rate[u], δc_uv(t_i, t_i + T_p)} + t_i
15:         alt :=max{rate[v], δc_uv(t_i, t_i + t_k^end[v])}
16:         if alt ≥ rate[v]:  // A path with higher transmission rate
17:            rate[v] := alt
18:            pre[v] := u
19:         end if
20:   P := empty sequence
21:   while pre[j] is defined do   //Construct the alternating path
22:      insert j at the beginning of P
23:      δc_{j,pre[j]}(t_i, t_i + t_k^end[j]) := 0
24:      j := pre[j]   // Traverse from destination to source
25:   if t^end[v]<d_k Return P
26:   if t^end[v]>d_k   //P cannot transmit f_k^I before its deadline
27:      split f_k^I into f_k1, f_k2
28:      Return P, f_k1, f_k2
29:   end if
```

If P cannot finish f_k^I's transmission before its deadline and f_k^I is split into f_{k1} and f_{k2}, we will use Algorithm 1 to identify an alternating path for f_{k2}. If the new identified alternating path cannot transmit f_{k2} before its deadline, f_{k2} is further split into two parts: f_{k2}^1 and f_{k2}^2. f_{k2}^1 is transmitted on the new alternating path and f_{k2}^2 is transmitted on e_{ij} by increasing the charging volume on e_{ij}. The new charging volume $\hat{v}_{ij}(t_i)$ at time t_i is calculated by:

$$\hat{v}_{ij}(t_i) = (\mathcal{A} - s_k + s(f_{k2}^2) \times T_r/(d_k - t_i). \tag{4.10}$$

Where $s(f_{k2}^2)$ is the size of f_{k2}^2. The flow start time and completion time of all flows on link e_{ij} will then be updated based on Equation (4.5), and the schedule table $S(t_i) =< f_k, S, D, t_k^{start} >$ will also be updated.

4.2.4 Forwarding Subflows with Rate Limiters

When sending video flows in the inter-datacenter network, we need to control the transmission rate so that the flows are sent out by using the links' available bandwidth capacities. Each datacenter i deploys a scheduler C_i to organize the sending queue of video flows, which attaches labels to all packets of each flow that record their corresponding transmission paths. For each video flow, the datacenter also uses a rate limiter [12, 97] to control its sending rate within the available bandwidth capacity, so that the links' current charging volume on the flow's transmission path will not increase. We will describe how to use rate limiter to split the packets of each flow in two different cases, i.e., DFs and IFs.

If flow f_k is a DF sending from datacenter i to datacenter j, all packets of f_k are transmitted using available bandwidth capacity on link e_{ij} (i.e., $\Delta c_{ij}(t_i, t_i + T_p)$). The number of packets transmitted per second r_k is calculated by:

$$r_k = \Delta c_{ij}(t_i, t_i + T_p)/\mu, \tag{4.11}$$

where μ is the size of a packet. The cloud provide specifies the value of μ, typically, μ is set to 1KB [58, 113].

If flow f_k is split into multiple subflows, e.g., f_k is split into f_k^D and f_k^I, and f_k^I is further split into f_{k1} and f_{k2}, the rate limiter needs to split the packets proportionally according to each link's available bandwidth capacity. Assume f_k is split into (f_{k1}, f_{k2},...,f_{km}), which are transmitted using links (e_{i1}, e_{i2},...,e_{im}). The total number of packets sent per second for f_k is:

$$r_k = \sum_{j=1}^{m} \Delta c_{ij}(t_i, t_i + T_p)/\mu. \tag{4.12}$$

In this way, the f_k's packets are proportionally distributed across all paths.

4.2.5 Setting Initial Charging Volume

According to the 95th percentile charging model, the charging volume at the beginning of the charging period is 0 and it increases gradually as the bandwidth usage goes up. Figure 4.4 shows an example of a link's charging volume increases over time. In this example, the charging volume

rises drastically in early time intervals and keeps relatively stable after a certain time interval. This property leads to a problem in our design. During early time intervals, as available bandwidth capacity of a direct link is 0 and can not transmit a video flow f_k on this path, f_k then needs to look for an alternating path. f_k cannot find any alternating path due to the reason that available bandwidth capacities of all links are 0. Finally, the direct link needs to increase its charging volume in order to transmit f_k. This process leads to extra scheduling latency due to insufficient available bandwidth capacities in all links.

To reduce the scheduling latency and make the schedule process simple during early time intervals of a charging period, we set an initial charging volume on each link, as shown in the dash line in Figure 4.4. Assume each charing period is divided into a number of time points $< t_0, t_1, ...t_{end} >$, i.e., it starts from time t_0 and ends at time t_{end}. The initial charging volume on each link e_{ij} at time t_0 is denoted by $\hat{v}_{ij}^0(t_0)$. $\hat{v}_{ij}^0(t_0)$ should be set properly. If $\hat{v}_{ij}^0(t_0)$ is too small, the scheduling latency cannot be reduced since a direct link does not have enough available bandwidth capacity to transmit flow f_k, and also we can hardly find an alternating path because other links' initial charging volumes are small. On the other hand, if $\hat{v}_{ij}^0(t_0)$ is too large, the initial charging volume is not fully utilized throughout the charging period, and the proposed scheduling scheme will not be effective in reducing each link's bandwidth cost. In this section, we introduce how to set the initial charging volume on each link based on the historical data, i.e., actual charging volume at the end of the last charging period (denoted by t_{end}), since it can be an indicator of how much traffic volume will be transmitted during current charging period.

We consider two factors in setting $\hat{v}_{ij}^0(t_0)$, which are the actual charging volume at time t_{end} on link e_{ij} ($\hat{v}_{ij}(t_{end})$) and the average actual charging volume at time t_{end} on all links in the inter-datacenter network (denoted by $\bar{V}(t_{end})$). $\bar{V}(t_{end})$ is calculated as:

$$\bar{V}(t_{end}) = \sum_{e_{ij} \in E} \hat{v}_{ij}(t_{end})/2|E|. \tag{4.13}$$

$\hat{v}_{ij}(t_{end})$ plays a major role in determining the initial charging volume on e_{ij}, because EcoFlow encourages sending video flows using direct link e_{ij} to reduce transmission time. As introduced in Section 4.2.1, $\hat{v}_{ij}(t_{end})$ can be calculated based on the actual historical traffic volume during previous charging period. We consider $\bar{V}(t_{end})$ because EcoFlow aims to transmit video flows using available bandwidth capacities of all links in the inter-datacenter network so as to control the charging volume

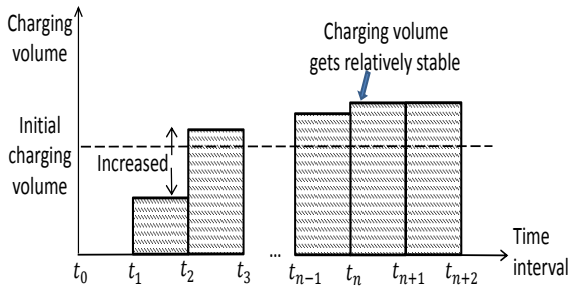


Figure 4.4: An example of setting an initial charging volume at the beginning of a charging period.

on a specific link and offload partial traffic to alternating paths. Combining these two factors, we calculate $\hat{v}_{ij}^0(t_0)$ by:

$$\hat{v}_{ij}^0(t_0) = \phi\hat{v}_{ij}(t_{end}) + \varphi\bar{V}(t_{end}). \tag{4.14}$$

$\phi \in (0,1)$ is a weight placed on $\hat{v}_{ij}(t_{end})$; $\varphi \in (0,1)$ is a weight placed on $\bar{V}(t_{end})$. At the beginning of each charging period, charging volume on link e_{ij} is set to $\hat{v}_{ij}^0(t_0)$. In Section 4.3.2, we will evaluate the performance of this strategy by setting different values of ϕ and φ.

4.2.6 Deal with Prediction Errors and Lack of Prior Knowledge

Our design is based on prediction. That is, we estimate the available bandwidth capacity on each link and utilize it to reroute videos. When there are prediction errors, e.g., there are more video flows waiting to be transmitted on a link than the estimated value because we overestimate the available bandwidth capacity, excessive number of videos will be waiting in the sending queue. To handle the prediction errors, our design can adapt to these prediction errors by revising the transmission schedule. When there are an excessive number of pending videos in the sending queue and EcoFlow can not find alternating paths for these videos, it then uses Equation (4.10) to calculate a new charging volume on this link.

When we lack prior knowledge of the charging volume during previous charging periods, as describe in Section 4.2.5, the cloud service provider will set each link's initial charging volume based on how much bandwidth cost it is willing to pay, or it can set the initial charging volume to 0 and let the charging volume increases gradually as more videos are transmitted.

4.2.7 Centralized Implementation of EcoFlow

In this section, we will put all components of EcoFlow together and describe the centralized implementation of EcoFlow. As mentioned in Section 4.2.2, each scheduler maintains a sending queue $Q(t_i)$=(<f_1, d_1, s_1>, <f_2, d_2, s_2>,...,<f_m, d_m, s_m>) to store all pending video flows on each link e_{ij} at time t_i, and the flows are sorted in ascending order based on their deadlines. The goal of EcoFlow is to calculate a schedule table $S(t_i)$. That is, it decides a transmission path for each flow, and splits a flow into subflows and identifies alternating paths for them if a flow is estimated to miss its transmission deadline. We describe the process of scheduling video flows on link e_{ij} in Algorithm 2. If t_i is the beginning of a charging period, the scheduler first sets an initial charging volume on e_{ij} (Line 3-5). It then calculates e_{ij}'s available bandwidth capacity (Line 6). For each f_k in the sending queue, EcoFlow calculates expected transmission time t_k^{end}. If t_k^{end} is earlier than f_k's transmission deadline, f_k will be transmitted on link e_{ij} (Line 9-10). Otherwise, EcoFlow splits f_k into f_k^D and f_k^I and identifies an alternating path for f_k^I (Lines 12-13). If an alternating path P is found to transmit f_k^I, we update the available bandwidth capacity on each edge of P (Line 14). If no alternating path can be found to transmit the whole volume of f_k^I, we increase charging volume on e_{ij} at time t_i according to Equation (4.10) (Lines 15-17). Finally, the scheduling table $S(t_i)$ is updated based on the scheduling results (Line 20).

Algorithm 2: Pseudocode for scheduling video flows on link e_{ij}.

```
1:  Input: G = (V, E); Q(t_i);
2:  Output: schedule table S(t_i) for all flows in Q(t_i)
3:    if t_i is the beginning of a charging period
4:      set initial charging volume v̂_ij^0(t_0)
5:    end if
6:    calculate available bandwidth capacity ṽ_ij(t_i, t_i + T_p)
7:    for each f_k ∈ Q(t_i)
8:      calculate expected transmission time t_k^end
9:      if t_k^end is smaller than deadline d_k
10:       f_k is transmitted directly on link e_ij
11:     else
12:       split f_k into subflows: f_k^D and f_k^I
13:       find alternating path P for f_k^I using Algorithm 1
14:       update available bandwidth capacity on each link of P
15:       if P cannot transmit whole volume of f_k^I
16:         increase v̂_ij(t_i) on e_ij according to Equation (4.10)
17:       end if
18:     end if
19:   end for
20:   update scheduling table S(t_i) for f_k and subflows
```

4.2.8 Distributed Implementation of EcoFlow

In order to prevent the single point of failure problem, we propose a distributed implementation of EcoFlow. As mentioned before, each datacenter i has a scheduler C_i. For flow scheduling on link e_{ij}, we select a scheduler on datacenter i or j as a master scheduler, denoted by scheduler C_{ij}, and the other schedule then becomes the slave scheduler. Scheduler C_{ij} is responsible for scheduling transmission of flows on e_{ij}, calculating the available bandwidth capacity (Δc_{ij}) on link e_{ij} using the same technique as in Section 4.2.1, and broadcasts this information to all schedulers in the network for alternating path identification.

At each time interval T_r, scheduler C_i and scheduler C_j report information of its pending flows on link e_{ij} (including flow ID, size and deadline) to scheduler C_{ij}. Scheduler C_{ij} then orders the flows by their deadlines in ascending order, and calculates start time and completion time for each flow using the same technique in Section 4.2.8. All flows on link e_{ij} are divided into DFs ($F^D(t_i)$) and IFs ($F^I(t_i)$). Scheduler C_{ij} builds a schedule table $S(t_i) =< f_k, S, D, t_k^{start} >$ for flows in $F^D(t_i)$ and forwards it to both scheduler C_i and scheduler C_j. As DFs can be transmitted directly through e_{ij}, scheduler C_i and scheduler C_j transfer flows in DFs according to the schedule table. Scheduler C_{ij} also need to find the alternating paths for $F^I(t_i)$ and notifies scheduler C_i and scheduler C_j the alternating paths.

Assume f_k^I is an IF flow from datacenter i to datacenter j, we need to identify an alternating path that can transfer f_k^I before its deadline. Due to lack of a centralized scheduler, the challenge of the distributed identification method lies in finding an alternating path through the cooperation of multiple schedulers. Under this scenario, identifying a path with the minimum transmission time f_k^I is complicated, so we aim to find a path that can transfer f_k^I before its deadline. As each datacenter has direct links connected to all other datacenters, sufficient number of datacenters can be chosen as relay datacenters in the alternating paths. With a high probability, we can find a relay datacenter and build a 2-hop alternating path which can transfer f_k^I before its deadline. While identifying a multi-hop (more than 2-hop) alternating path requires cooperation of more schedulers and is not efficient in time complexity, in order to simply the implementation and achieve algorithm time efficiency, we identify a 2-hop alternating path $P = (i, h, j)$ for f_k^I in our proposed method, that is, find an intermediate datacenter h and transfer f_k^I on path $P = (i, h, j)$. Multiple candidate datacenters might be able to relay and transfer f_k^I before its deadline, we then contact

each datacenter's scheduler and randomly choose a datacenter h who are able to relay f_k^I as the intermediate datacenter. Note that this routing path identification method can be extended to identify alternating paths with more than two hops.

The information of available bandwidth capacity on each link at each time interval is shared among all schedulers by broadcasting. The intermediate datacenter h is selected according to Algorithm 4. In this algorithm, we try each datacenters in $V \backslash j$ to build a candidate path (Line 3). For each candidate path, we then calculate f_k^I's transmission time on this path (Line 4). If path P can transfer f_k^I before its deadline, we further check if the links on P have extra bandwidth capacities (Line 5-10). When intermediate datacenter h is found, scheduler C_i then forwards f_k^I to datacenter h, and scheduler C_h further forwards it to its destination j. If scheduler C_i cannot find a transit datacenter h, scheduler C_i increases the charging volume on e_{ij} according to Equation (4.10).

Algorithm 3: Pseudocode for finding intermediate datacenter h.

```
1:  Input:  δc_ij(t_i, t_i + T_p),  ∀ij ∈ E;
2:  Output: intermediate datacenter h between source i to destination j
3:    for each  vertex q in V\j:
4:        t_k^end:=s_k/ min{Δc_iq, Δc_qj} + t_i   //Calculate completion time
5:        if t_k^end < d_k:  //Guarantee the transmission deadline
6:           scheduler C_i contacts scheduler C_q
7:           if δc_iq(t_i, t_i + t_k^end) > 0 and δc_qj(t_i, t_i + t_k^end) > 0:
8:                h := q
9:           end if
10:       end if
11:   end for
```

4.3 Performance Evaluation

We conducted experiments on the PlanetLab [86] real-world testbed and Amazon EC2 platform [39] to evaluate the performance of EcoFlow in comparison with other systems. For EcoFlow, we tested both the implementation with a centralized scheduler (denoted as EcoFlow-C) and distributed implementation (denoted as EcoFlow-D). We compare the performance of EcoFlow with three datacenter traffic scheduling strategies: 1) Direct transfer (denoted as Direct), which directly transfers video flows to the destination whenever the video transfer requests are initiated by the cloud provider without considering each link's charging volume; 2) JetWay [45], which transfers video flows whenever the video transfer requests are initiated by the cloud provider, at a rate calculated by its size divided by its corresponding maximum tolerable transfer time. When a video flow is expected to increase a link's current charging volume, it splits the video flow into two sub-flows, and

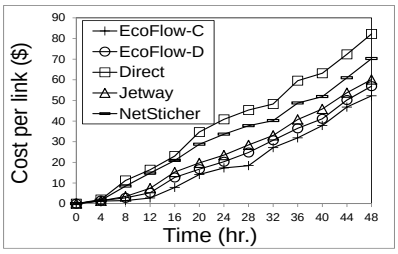
the sub-flows are transmitted along alternating paths to utilize the available bandwidth capacity of each link; and 3) NetSticher [81], which is a *store-and-forward* method. NetSticher transfers delay-tolerant data between two datacenters only when both datacenters are in off-peak hours. When there are no common off-peak hours between both datacenters, an intermediate datacenter is used to store the data temporarily and then forward it to the destination datacenter. In both PlanetLab and EC2 experiments, we defined two types of videos: Standard Definition (SD) videos with sizes randomly selected in [500, 800] MB, and High Definition (HD) videos with sizes randomly selected in [2, 4] GB [45]. We assumed that the traffic load for each datacenter displays a periodic diurnal pattern [81]. For simplicity, we further assumed that 10-12am and 6pm-12am of a node's local time are peak hours. A datacenter transfers x and y videos per hour (including both SD and HD videos) to all other datacenters during its peak hours and off-peak hours, respectively, where x and y were randomly selected from [2, 5] and [0, 1], respectively. The transfer request of each video is initiated at a random time during the selected hours, and its deadline is chosen in [30, 120] minutes after the transfer request's initiated time. We assumed a video with maximum tolerable transfer time longer than 60 minutes to be a delay-tolerant. We set $T_p = 1$ hour and $T_r = 5$ minutes. We simulated an inter-datacenter network running for 48 hours for all methods. We set this 48 hour period as an independent charging period and calculated the bandwidth cost on each link at the end of the experiment. In EcoFlow-C and EcoFlow-D, we had a 48 hour warmup period and used the traffic records in this period to predict the traffic volume on each link during the charging period. We also set the initial charging volume based on the charging volume in this warmup period according to Section 4.2.5. We calculated the bandwidth costs under the 95th percentile charging model.

4.3.1 Experimental Results for Overall Performance

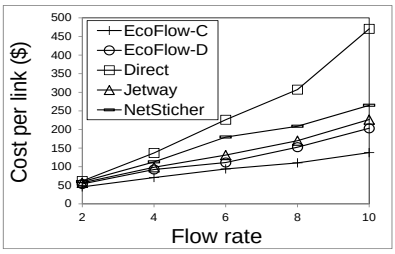
We first present the overall performance of EcoFlow in terms of bandwidth cost, percentage of flows transmitted within the charging volume and percentage of transferred flows within deadlines. In order to compare EcoFlow with other scheduling methods, we set the initial charing volume to 0 in these experiments.

4.3.1.1 Experiments on PlanetLab

We used 15 distributed nodes worldwide to simulate 15 datacenters, including 7 nodes in North America, 5 nodes in East Asia and 3 nodes in Europe. On each link between two datacenters,

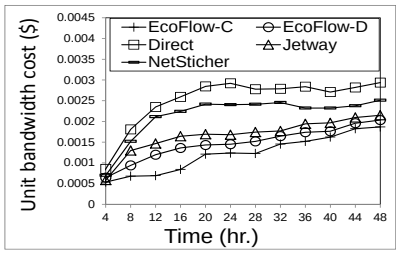


(a) Results at different time intervals.

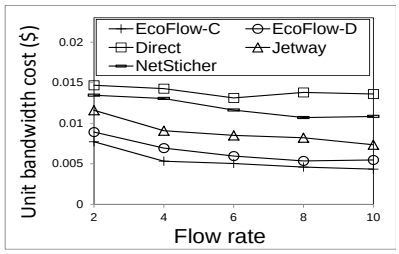


(b) Results at different flow rates.

Figure 4.5: Average bandwidth cost per link on PlanetLab.



(a) Results at different time intervals.

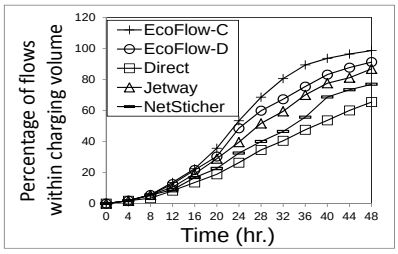


(b) Results at different flow rates.

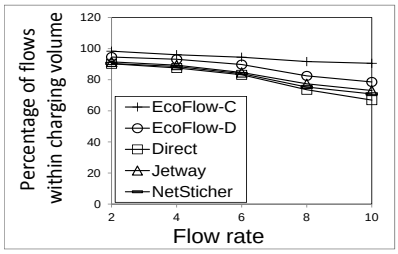
Figure 4.6: Average unit bandwidth cost on PlanetLab.

the bandwidth capacity is randomly selected in [10, 600] MB, and the bandwidth cost per unit (MB) is randomly selected in [50, 400] [45]. Each node's time zone is determined based on its location. In the experiment, we used the TCP protocol to transfer data between different nodes.

We first defined a metric of *bandwidth cost per link* as the sum of bandwidth payment cost on all links divided by the total number of links in the network. Figure 4.5(a) shows the average bandwidth cost per link at each time interval. We see that as time evolves, bandwidth payment cost for each method is increasing due to the reason that bandwidth payment cost is a function of how long the link's bandwidth is used according to Equation (4.1). The result also follows: EcoFlow-C<EcoFlow-D<JetWay<NetSticher<Direct. Direct results in the highest bandwidth cost. When a video transfer request arrives at the source datacenter, it immediately transfers the video by using only the direct link between two datacenters without considering the current charging volume on the link. NetSticher postpones the transmission of delay-tolerant videos until both source datacenter and destination datacenter are during off-peak hours, so that the traffic load during peak hours is alleviated and it generates less average bandwidth cost than Direct. However, as the available bandwidth capacity is not fully utilized during peak hours, there is still room for NetSticher to further reduce the bandwidth cost. JetWay is able to incur less bandwidth cost than NetSticher



(a) Results at different time intervals.



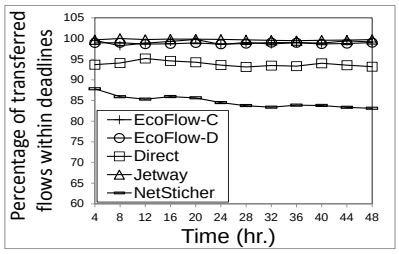
(b) Results at different flow rates.

Figure 4.7: Average percentage of flows transmitted within the charging volume on PlanetLab.

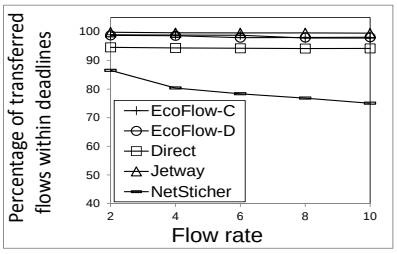
by controlling the transmissions of current videos within the charging volume. Also, when a video is expected to increase a link's charging volume, it splits the video into subflows and reroutes the subflows to links that are under-utilized. However, as videos are transmitted immediately when the video transfer requests are initiated by the cloud provider, the bandwidth cost will increase when a large number of video transfer requests arrive simultaneously. EcoFlow generates the least bandwidth cost among all comparison methods. EcoFlow schedules the flows on each link based on their deadline tightness, and postpones the transmission of video flows to make the current traffic within the charging volume. Flows that are expected to miss their deadlines are splitted into subflows, which will be rerouted to alternate paths that are constructed by under-utilized links. Also, EcoFlow transmits each video with the link's available bandwidth capacity, so that the charging volume is fully utilized. Note that EcoFlow-C performs better than EcoFlow-D as it gains full knowledge of all under-utilized links in the network, and thus has higher probability to identify a reroute path for IFs using under-utilized links.

Next, in order to test the performance of each method at presence of different traffic loads, we changed the flow arrival rates during a link's peak hours from 2 to 10 flows per hour on each link. Figure 4.5(b) shows the bandwidth cost per link at the end of the 48-hour charging period at different flow rates. We see that as more flow transmission requests are initiated hourly, the average cost per link tends to increase. This is because when a larger number of videos need to transfer between datacenters during the charging period, more bandwidth is generally required on each link to transmit all videos, so the charging volume on each link increases. The relative performance of different methods in Figure 4.5(b) concurs with that in Figure 4.5(a) due to the same reason.

We then present a performance metric of *average unit bandwidth cost*, which is defined as the sum of bandwidth payment cost on all links divided by the total volume of video flows (MB)



(a) Results at different time interval.



(b) Results at different flow rates.

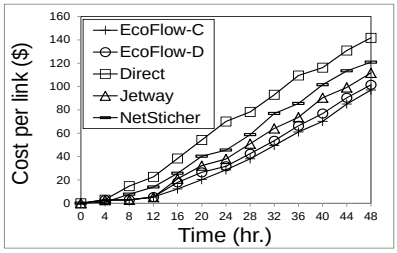
Figure 4.8: Percentage of transferred flows within deadlines on PlanetLab.

transmitted in the network. An effective scheduling system should be able to reduce the average unit bandwidth cost, i.e., use the same bandwidth cost to transmit a larger size of videos. Figure 4.6(a) plots average unit bandwidth cost at each time interval. We see that as time evolves, the average unit bandwidth cost for all methods generally increases because bandwidth payment cost is increased as explained in Figure 4.5(a). The relative performance between different methods follows: EcoFlow-C<EcoFlow-D<JetWay<NetSticher<Direct. EcoFlow generates the unit bandwidth cost among all methods as it postpones the transmission of video flows to make the current traffic within the charging volume, and it splits the flows that are expected to miss their deadlines into suflows and utilizes other links' available bandwidth capacities to reroute these subflows. Thus, EcoFlow can efficiently reduce the average unit bandwidth cost.

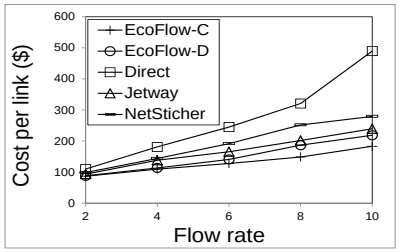
As in Figure 4.5(b), we changed the flow arrival rates during a link's peak hours from 2 to 10 flows per hour and tested EcoFlow's performance with respect to average unit bandwidth cost. Figure 4.6(b) plots the average unit bandwidth cost at the end of the 48-hour charging period at different flow rates. We see that the average unit bandwidth cost generally drops when the flow arrival rate increases from 2 to 8 flows per hour, and it then keeps stable when the flow arrival rate increases from 8 to 10 flows per hour. This is due to the reason that when a larger number of videos are transmitted between datacenters during the charging period, the links connected the datacenters have higher utilization and more videos are sent by using current charging volume, which reduces the bandwidth payment cost per video unit. However, when the flow arrival rate reaches a specific point (8 flows per hour in this case), the links' available bandwidth capacities are overutilized and they need to increase the charging volumes in order to transmit higher rates of video flows. Thus, the average unit bandwidth cost keeps stable. The relative performance of different methods in Figure 4.6(b) concurs with that in Figure 4.5(b) due to the same reason.

Link Cap. (Mbps)/Cost	North California	Oregon	Virginia	Sao Paulo	Ireland	Singapore	Tokyo
North California	–	520.40/1	252.67/2	116.75/15	98.67/20	103.69/27	173.06/30
Oregon	545.06/1	–	215.84/3	81.18/17	104.22/15	81.99/25	152.75/27
Virginia	240.78/2	210.64/3	–	139.41/10	221.55/10	81.44/15	110.10/17
Sao Paulo	40.98/15	60.84/17	11.85/10	–	22.41/25	9.59/18	62.42/22
Ireland	106.45/20	135.02/15	215.85/10	90.12/25	–	77.89/23	76.10/20
Singapore	124.40/27	110.95/25	84.54/15	57.31/18	80.92/23	–	242.80/5
Tokyo	178.36/30	143.44/27	99.40/17	61.99/22	43.61/20	116.33/5	–

Table 4.2: Link Capacities and Costs per Traffic Unit in the Amazon EC2 Inter-Datacenter Network [45]



(a) Results at different time intervals.

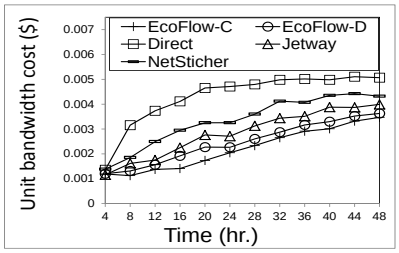


(b) Results at different flow rates.

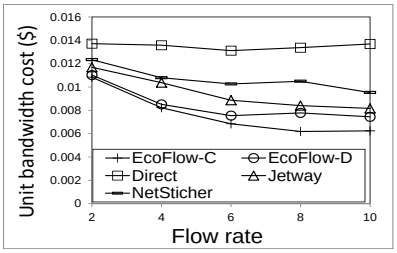
Figure 4.9: Average bandwidth cost per link on EC2.

Figure 4.7(a) shows the average percentage of flows transmitted within the current charging volume at different time intervals. If a large portion of the flows are transmitted by utilizing the current charging volume, a link's charging volume will not further increase. An effective flow scheduler should provide a large percentage of flows transmitted within the current charging volume, so that the bandwidth cost during current time interval will not further increases. We see that performance of different methods with respect to average percentage of flows transmitted within the charging volume follows: EcoFlow-C>EcoFlow-D>JetWay>NetSticher>Direct. In JetWay and Direct, if a large number of video transfer requests arrive at a specific time interval, the videos will be transmitted immediately. And these videos are likely to result in high bandwidth usage at the time interval and increase the charging volume. NetSticher performs transmissions of delay-tolerant videos only during off-peak times, the charging volume is likely to increase when a large number of non-delay-tolerant videos are transmitted during the peak hours. EcoFlow-C and EcoFlow-D aim to transfer flows within the charging volume by postponing the transmission of flows with late deadlines, thus yield the highest percentage of flows transmitted within the charging volume.

Figure 4.7(b) shows the average percentage of flows transmitted within the charging volume at different flow rates. We see that as more videos need to transfer between datacenters hourly, smaller percentage of flows can be transmitted within the charging volume, i.e., the charging volumes on all links need to increase in order to accommodate higher flow rates. This is due to the reason

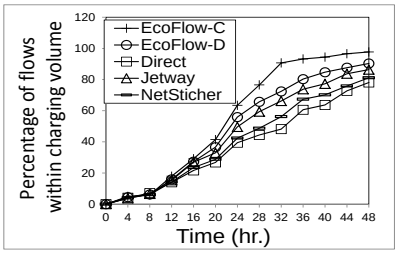


(a) Results at different time intervals.

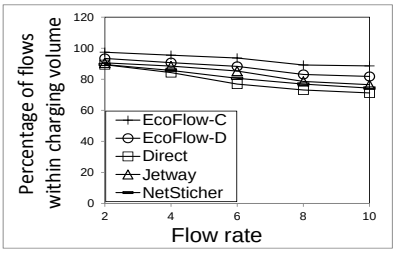


(b) Results at different flow rates.

Figure 4.10: Average unit bandwidth cost on EC2.

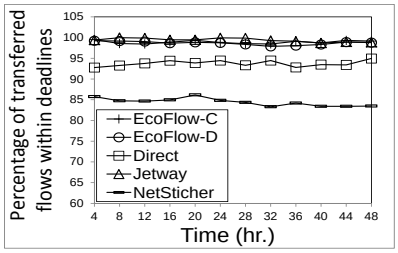


(a) Results at different time intervals.

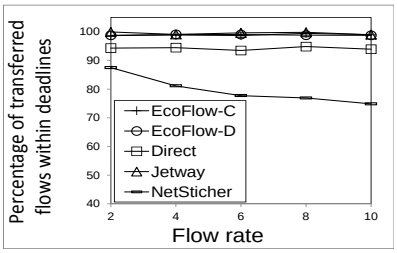


(b) Results at different flow rates.

Figure 4.11: Average percentage of flows transmitted within the charging volume on EC2.



(a) Results at different time interval.



(b) Results at different flow rates.

Figure 4.12: Percentage of transferred flows within deadlines on EC2.

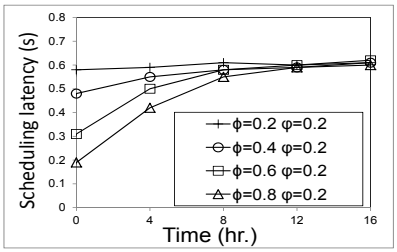
that increased bandwidth is needed to transfer a large number of videos during each time interval and thus likely to increase the charging volume. As a result, larger percentage of flows are likely to be transmitted by increased charging volume on the links. The relative performance of different methods mirrors that in Figure 4.7(a) due to the same reason.

Figure 4.8(a) and Figure 4.8(b) show the percentage of video flows that are transferred within their deadlines across different time intervals and at different flow rates, respectively. We see that the result follows: JetWay>EcoFlow-C>EcoFlow-D>Direct >NetSticher. NetSticher provides the least percentage of transferred videos within the deadlines due to the reason that it postpones the transmission of delay-tolerant videos from peak hours to off-peak hours, and if a link's available bandwidth capacity during the off-peak hours is not enough to transfer all waiting videos postponed from peak hours, a number of videos are likely to miss their transmission deadlines. Direct produces higher percentage of transferred videos within the deadlines than NetSticher, due to the reason that a video begins transmission whenever the transfer request arrives at the source datacenter. EcoFlow and JetWay generate comparably high percentage of transferred videos within the deadlines, as they both consider a video's transmission deadline when scheduling the video's transmission and aims to use the available bandwidth capacities from all links to finish the video's transmission before its deadline.

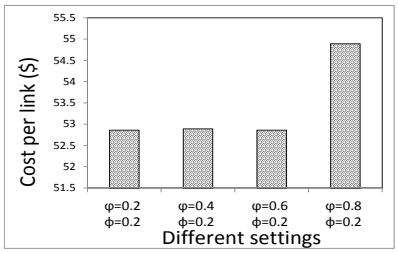
4.3.1.2 Experiments on EC2

We have conducted our experiments in the Amazon EC2 platform, which is one of the dominant Infrastructure as a Service (IaaS) cloud providers. There are a total of 7 datacenters on EC2, the capacity and cost per traffic unit of each link are set according to the studies in [45]. The settings are shown in Table 4.2. We assigned the diurnal load described above to each datacenter based on the time zone it resides in.

Figure 4.9(a) shows the average cost per link at different time intervals for the 95th percentile charging model. We see that the per link cost follows: EcoFlow-C<EcoFlow-D<JetWay<NetSticher<Direct due to the same reason in Figure 4.5(a). Figure 4.9(b) shows the average bandwidth cost per link at the end of the 48-hour charging period under different flow rates. We obtain the same observation as that in Figure 4.5(b) due to the same reason, i.e., more bandwidth is generally required on each link to transmit a larger number of videos during a specific charging period, so the charging volume on each link increases as well.



(a) Average scheduling latency.



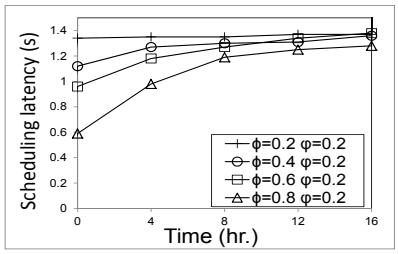
(b) Average bandwidth cost per link.

Figure 4.13: Effectiveness of setting initial charging volume in EcoFlow-C on PlanetLab.

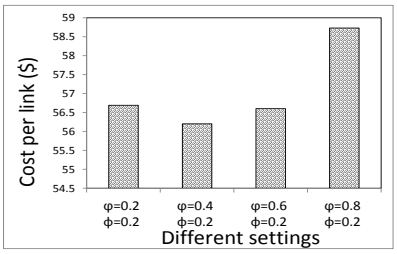
Figure 4.10(a) shows the average unit bandwidth cost at different time intervals under the 95th percentile charging model. We see that the relative performance between different methods follows: EcoFlow-C<EcoFlow-D<JetWay<NetSticher<Direct due to the same reason as in Figure 4.6(a). Figure 4.10(b) shows the average bandwidth cost per link at the end of the 48-hour charging period under different flow rates. We obtain the same observation as that in Figure 4.6(b) due to the same reason.

Figure 4.11(a) shows the average percentage of flows transmitted within the charging volume at different time intervals. JetWay and Direct do not take advantage of temporal features of the video traffic on each link, they both transfer the video immediately after a transfer request arrives. Thus, during peak hours, a datacenter may transfer more traffic than the charging volume; while in off-peak hours, the transmission of small volume of traffic results in the under-utilization of links. EcoFlow-C and EcoFlow-D both yield a high percentage of flows transmitted within the charging volume, due to the reason that when the charging volume is fully utilized by some emergent flows currently, they postpone the transmission of some delay-tolerant video flows to off-peak hours when the links are light of traffic. Figure 4.11(b) shows the average percentage of flows transmitted within the charging volume at different flow rates. We see that a smaller percentage of flows can be transmitted within the charging volume as flow rate increases. This is due to the same reason as in Figure 4.7(b).

Figure 4.12(a) and Figure 4.12(b) show the percentage of video flows that are transferred within their deadlines across different time intervals and at different flow rates, respectively. We see that the result follows: JetWay>EcoFlow-C>EcoFlow-D>Direct >NetSticher due to the same reason as in Figure 4.8(a) and Figure 4.8(b) .

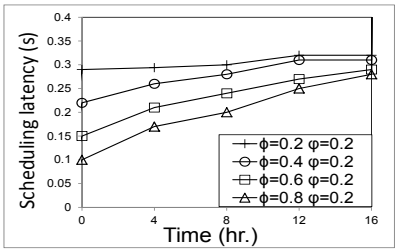


(a) Average scheduling latency.

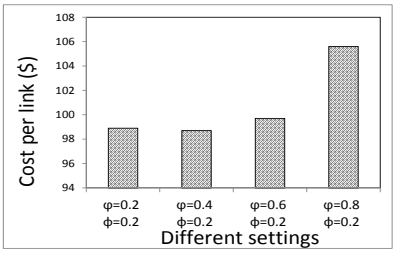


(b) Average bandwidth cost per link.

Figure 4.14: Effectiveness of setting initial charging volume in EcoFlow-D on PlanetLab.

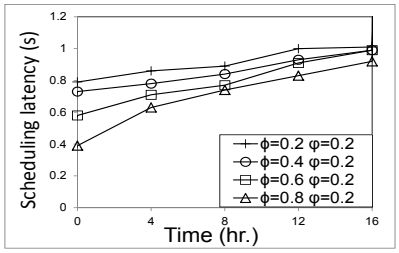


(a) Average scheduling latency.

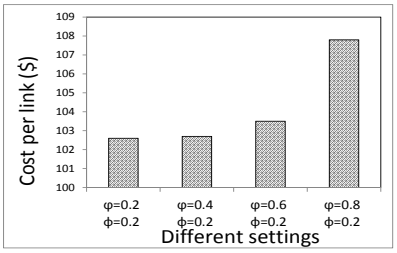


(b) Average bandwidth cost per link.

Figure 4.15: Effectiveness of setting initial charging volume in EcoFlow-C on EC2.



(a) Average scheduling latency.



(b) Average bandwidth cost per link.

Figure 4.16: Effectiveness of setting initial charging volume in EcoFlow-D on EC2.

4.3.2 Effectiveness of Setting Initial Charging Volume

In this section, we tested the performance of both EcoFlow-C and EcoFlow-D when we set an initial charging volume at the beginning of the charging period according to Section 4.2.5. We set φ in Equation (4.14) to a fixed value of 0.2, and varied the value of ϕ from 0.2 to 0.8.

4.3.2.1 Experimental Results on PlanetLab

Since the purpose of setting an initial charging volume is to reduce the scheduling latency during early time intervals of a charging period, we first present the average scheduling latency on all links. It is the time span from when a scheduler receives a sending queue of videos on a link until the time when the scheduler finished the scheduling of these video and updating the schedule table. Figure 4.13(a) shows the average scheduling latency of EcoFlow-C from the 0th to the 16th hour of the charging period. We see that the scheduling latency gradually increases with time. This is because when few videos are pending at early time intervals, a large portion of videos can be transmitted directly by using available capacities of direct links, so the scheduling latency is short since the scheduler does not need to search alternating paths for these videos. As more video flows are transmitted in the network, available bandwidth capacities of some links are used up and videos on these links need to be split and rerouted to other links, so the scheduler needs longer latency to update the scheduling table. We also see that larger value of ϕ leads to shorter scheduling latency. According to Equation (4.14), large value of ϕ leads to large initial charging volume and high available bandwidth capacities on the links. The scheduling latency is short because most videos can be transmitted through direct links. On the other hand, small value of ϕ leads to longer scheduling latency as the scheduler needs to search alternating paths for some videos that are not able to be transmitted on direct links.

The negative effect of large value of ϕ is that it may lead to underutilization of the initial charging volume, and the bandwidth cost is not minimized at the end of the charging period. To further evaluate the effect of initial charging volume in bandwidth cost reduction, we then plot average bandwidth cost per link at the end of the 48 hour charging period in Figure 4.13(b). We see that higher value of ϕ generally leads to higher bandwidth cost due to the reason that smaller charging volume may be adequate in transmitting all video flows. Therefore, it is important to determine an appropriate initial charging volume to reduce scheduling latency and meanwhile constrain bandwidth

cost.

We then evaluate the effectiveness of setting initial charging volume in EcoFlow-D. Figure 4.14(a) shows the average scheduling latency of EcoFlow-D from the 0th to 16th hour of the charging period. Compared to Figure 4.13(a), we observe that EcoFlow-D generates higher scheduling latency due to the reason that in EcoFlow-D, each datacenter has its own scheduler, and schedulers need to communicate with each other in order to search alternating paths for indirect flows. Figure 4.14(b) shows average bandwidth cost per link for EcoFlow-D at the end of the 48 hour charging period. We see that the experimental results concur with that in Figure 4.13(b) due to the same reason.

4.3.2.2 Experimental Results on EC2

We also evaluate the effectiveness of setting initial charging volume on EC2. Figure 4.15(a) shows the average scheduling latency of EcoFlow-C in different time intervals of the charging period. Compare to Figure 4.13(a), we observe that the scheduling latency is generally shorter on EC2 than on PlanetLab because there are more datacenters on PlanetLab, so EcoFlow-C needs longer latency to calculate the available bandwidth capacities of all links and search alternating paths for indirect video flows when scheduling video flows. Figure 4.15(b) shows the average bandwidth cost per link for EcoFlow-C. We see that higher value of initial charging volume leads to higher bandwidth cost because the initial charging volume can be larger than the actual charging volume needed to transmit all video flows. The relative performance between different settings of ϕ in both Figure 4.15(a) and Figure 4.15(b) concurs with that in Figure 4.13(a) and Figure 4.13(b) due to the same reason.

Figure 4.16(a) plots the average scheduling latency of EcoFlow-D in different time intervals of the charging period. Compared to Figure 4.14(a), we see that EcoFlow-D generates shorter scheduling latency on EC2 than on PlanetLab due to the reason that EC2 has fewer datacenters and the scheduling complexity is smaller. Figure 4.16(b) shows average bandwidth cost per link for EcoFlow-D at the end of the 48 hour charging period. We see that the results in Figure 4.16(a) and Figure 4.16(b) concur with those in Figure 4.14(a) and Figure 4.14(b) due to the same reason.

Chapter 5

EAFR: An Energy-Efficient Adaptive File Replication System In Cloud Storage System

In this chapter, we introduce an Energy-Efficient Adaptive File Replication System (EAFR). We first introduce the background of file replication in data-intensive clusters. We then introduce our proposed EAFR system in detail. Experimental results on a real-world cluster show the effectiveness of EAFR and proposed strategies in reducing file read latency, replication time, and power consumption in large clusters.

5.1 Overview

A typical cluster file system uses a hierarchical storage architecture, as shown in Figure 5.1. The bottom layer consists of a set of storage servers, where the files (aka objects or blocks) are stored. In order to guarantee data availability in face of network failure or hardware damage, cluster file system makes multiple replicas for each file [14]. A replication factor (r_i) and a fault-tolerance factor (π_i) for file (f_i) are predefined in the system, which ensure that each file f_i has r_i replicas and the replicas are distributed in more than ($\pi_i < r_i$) fault domains (i.e. racks). Typically, HDFS uses $r_i = 3$ and $\pi_i = 1$, i.e., each file is stored in three servers across at least two racks.

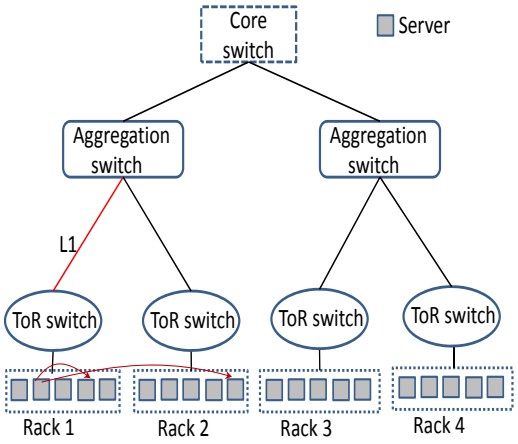


Figure 5.1: Architecture of hierarchical storage system.

When one rack suffers from a failure, the file is still available. The red arrows in Figure 5.1 shows an example of the distributed writes when storing a file with $r_i = 3$ and $\pi_i = 1$. By creating a constant number of replicas for each file, current replication systems neglect the heterogeneity in file popularity. Some hot files attract a large amount of read requests from clients, while some cold files attract few visits. As a result, copying files to only three servers is insufficient to meet the stringent response requirement for hot files but wastes resources for cold files.

On top of the servers are ToR switches, which are located within the Ethernet to aggregate the connectivity of all servers. The ToR switches maintain connections to the rest of network through aggregation switches, on top of which is the core switch. In this network architecture, the link capacity between switches is bounded by hardware limitations (e.g., NIC speed). Though link utilizations inside clusters are generally low and stable, there exist network congestions due to skewed link utilization [49]. For example, link L1 (marked in red) in Figure 5.1 may become a bottleneck if the there are many writes to Rack 1. Current replication policy does not consider link utilization when transmitting file replicas. Also, current clusters must keep all servers running constantly to guarantee file availability, which is very costly in power consumption. The files are skewed in popularity, and many files rarely get accessed during their lifetime [5]. Thus, we can save power and management expense by putting servers storing the cold files in a "powernap" state. We summarize the shortcomings of current file replication methods as follows:

- A fixed number of replicas for each file is insufficient to provide quick file read for hot files while wastes resources for storing replicas of cold files.

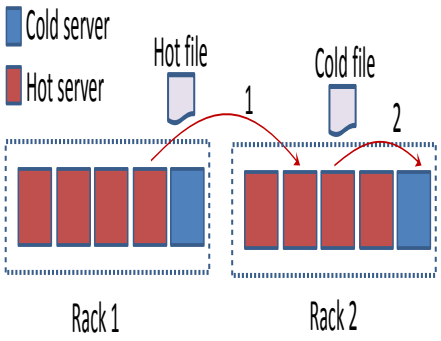


Figure 5.2: An overview of EAFR.

- Random selection of replica destinations requires keeping all servers active to ensure data availability, which however wastes power consumption.
- As the random selection of replica destinations does not consider destination bandwidth and request handling capacity, network congestions may occur due to capacity limitation of some links and server may become overloaded by data requests.

The goal of EAFR is to cope with these problems and provide an effective and energy-efficient file replication strategy. In this dissertation, if a file is striped into multiple blocks, we treat each block as an independent file.

5.2 System Design of EAFR

5.2.1 Energy-efficient and Popularity-adaptive File Replication

In large data-intensive clusters, most popular files are generally small in size, while large files seldom get read [41]. Therefore, replicating and migrating popular files is relatively light in storage and bandwidth cost. Taking advantage of this characteristic in clusters, EAFR increases the number of replicas of popular files in order to boost their availability and reduces the number of replicas of cold files in order to save resources. Figure 5.2 shows an overview of EAFR. EAFR divides servers into hot servers and cold servers: hot servers consume more power and provides prompt response for file requests; while cold servers stay in sleeping mode with 0% CPU utilization and low energy consumption. Each file f_i must have r replicas in all servers and $\epsilon < r$ replicas in hot servers, where r and ϵ are pre-defined numbers to guarantee file availability under server failures. For a file with

a high visit rate, EAFR creates an extra replica and places it to a hot server, which is shown in step 1. The new replica is used to balance the read requests of a hot file among servers where the file replicas are stored. For a file with a low visit rate (i.e., a cold file), EAFR reduces the number of replicas of the file in the hot servers if it is larger than ϵ. That is, a replica in a hot server is either deleted or migrated to a cold server in order to save the power consumption, which is shown in step 2. This operation does not affect the availability of the cold file as it rarely gets read. In the following, we introduce how to set hot servers and cold servers (Section 5.2.1.1), how to identify hot files and cold files based on their visit rates (Section 5.2.1.2), and the details of the energy-efficient and popularity-adaptive file replication algorithm (Section 5.2.1.3). Table 5.1 shows the notations used in this dissertation.

Table 5.1: Table of important notations.

f_i: File i
s_j: Server j
r_i: # of replicas
π_i: Fault-tolerance factor
a_{ij}: Replica j for f_i
v_i: Total # of reads for file f_i
v_{ij}: # of reads for replica a_{ij}
c_{c_j}: Service capacity of server j
b_j: Size of file j
ϕ_j: Remaining capacity of server j
c_{s_i}: Storage capacity of server i
∇: Remaining storage threshold
τ_u: Upper bound for total # of reads of hot file
σ_u: Upper bound for total # of reads of hot replica
τ_l: Lower bound for total # of reads of cold file
σ_l: Lower bound for total # of reads of cold replica
V_t: Transmission speed in time window t
w_{t_j}: Weight of selecting server j based on transmission rate
w_{r_j}: Weight of selecting server based on remaining capacity
p_j: Chance of selecting server j based on overall evaluation
c_b: Bandwidth capacity of the destination node
B_a: Proportion of available bandwidth
η_1: Highly utilized network capacity threshold
η_2: Under-utilized network capacity threshold
δ_i: Transmission rate adjust-up factor
β_i: Transmission rate adjust-down factor
ϵ: minimum # of replicas stored in hot servers

5.2.1.1 Different Types of Servers Based On Energy Consumption

Transitioning servers to an inactive, low power sleep /standby state (i.e., scale-down) is a technique to conserve energy. It trades energy consumption with server performance (e.g., CPU utilization). Table 5.2 shows the power consumption characteristics of the HP ProLiant ML110 G5 server at different server performances represented by server CPU utilization [19]. Higher CPU utilization consumes more power, and when a server runs at 0% CPU utilization (e.g., in sleeping state), the power consumption is 93.7Watts. In EAFR, we define three types of servers: hot servers, cold servers and standby servers. A hot server runs at the active state, i.e., with CPU utilization greater than 0. A cold server is in the sleeping state with 0 CPU utilization and inactive DRAM and disks and it does not serve any file read request. A standby server is a temporary hot server that will be transitioned to a cold server. To maintain the consistency of stored files, cold servers wake up periodically (e.g., once a week) and check for file consistency. When there is a rack failure or a server failure inside clusters, cold servers storing the lost files will be turned on and become hot servers.

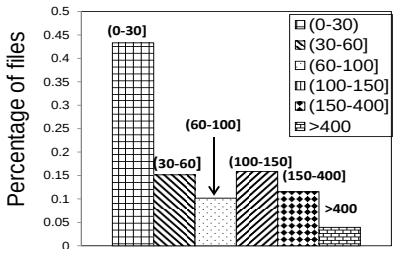
Table 5.2: Energy consumption for different CPU utilizations in Watts [19].

CPU Utilization	0%	20%	40%	60%	80%	100%
HP ProLiant G5	93.7	101	110	121	129	135
Server status	cold	hot	hot	hot	hot	hot

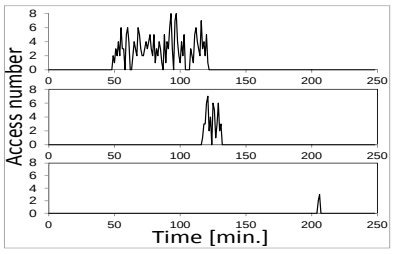
As a cold server runs at smaller power consumption compared to a hot server, switching hot servers to cold mode can save energy. A cold server stores cold files with few read requests. Writing a file to a cold server needs to wake up the server, which consumes more energy and may offset the benefit of sleeping [76]. Also, it creates excessive latency to transition a server from sleeping mode to active mode and thus delays the write operation. Therefore, we use a standby server to collect all cold files and turn into a cold server when its storage is full. A standby server still serves file requests as hot servers do.

5.2.1.2 Different Types of Files Based on Read Rates

In HDFS, more than 90% files exhibit a relatively short hotness lifespan (i.e., less than 3 days) and a significant portion of data is cold (i.e., never gets read) [67]. In order to justify the heterogeneity of file popularity in large clusters, we analyzed the file storage system trace data from Sandia National Laboratories [105], which records the number of file reads for 16,566 accessed files



(a) Percentage of files with different number of reads.



(b) The number of file reads over time.

Figure 5.3: Trace analysis on file read pattern.

during 4 hour run. Figure 5.3(a) shows the percentage of files attracting different range of file reads. We see that about 43% files receive less than 30 reads and 4% files receive a large number of reads (i.e., >400). The results confirm that most of these files attract a small amount of file reads and hence do not need many replicas. Popular files constitute a small percentage of files, thus will not generate a large overhead by creating more replicas. We sorted the files by their number of reads within a 4 hour period, then identified files with the 99th, 50th, and 25th percentiles and plotted their read count over time in Figure 5.3(b) from the top to the bottom, respectively. These figures demonstrate the variation in file access pattern for files with different popularities over time. We see that these files tend to attract a relatively stable number of reads within a short period of time. Thus, extra replicas can be created to meet the frequent short-term read requests for hot files, and then are deleted when they become cold. Inspired by the observations of the previous works and the above analysis, we can group files into different categories based on popularity and perform different operations according to their popularity. We present how to determine hot files and cold files below.

Current replication factor of f_i is r_i, and the replicas of f_i is denoted by vector $A_i = \{a_{i1}, a_{i2}, ...a_{ir_i}\}$. The number of reads for replica a_{ij} at time interval T is denoted by v_{ij}, and the total number of reads for file f_i is denoted by v_i, and

$$v_i = \sum_{j=1}^{r_i} v_{ij}. \tag{5.1}$$

First of all, a hot file should have a large number of concurrent reads across all its replicas. We define a hot file as a file with average read rate per replica exceeds a pre-defined threshold (τ_u):

$$v_i/r_i > \tau_u. \tag{5.2}$$

Secondly, we also consider the read rate of individual replicas. In locality-aware file reads in large clusters, clients read nearby replicas, so a large amount of concurrent reads for a file may be drawn by some replicas, which also reflects the popularity of the file. Therefore, a hot file may gain high read rate in some of its replicas. Thus, if more than a certain fraction (denoted by γ) of a file's replicas attract an excessive number of reads (denoted by σ_u), we consider this file a hot file:

$$\sum_{j=1}^{r_i} I(v_{ij} > \sigma_u) > r_i\gamma_v \ (0 < \gamma_v < 1), \tag{5.3}$$

where $I(\cdot)$ is an indication function, and $I(A)$=1 when the assertion A is satisfied. If either Equation (5.2) or Equation (5.3) is satisfied, file f_i is a hot file represented by $H(f_i) = 1$.

Similarly, we use Equation (5.4) and Equation (5.5) to determine if file f_i is a cold file. In the equations, τ_l is the lower bound of the number of reads per replica in T.

$$v_i/r_i < \tau_l. \tag{5.4}$$

Equation (5.4) shows that a cold file receives a small amount of concurrent reads across its replicas. A cold file waste storage space if the number of its replicas is large.

$$\sum_{j=1}^{r_i} I(v_{ij} < \sigma_l) > r_i\gamma_v \ (0 < \gamma_v < 1) \tag{5.5}$$

In Equation (5.5), σ_l denotes the lower bound for the number of reads that a file replica receives in T. If more than γ_v fraction of a file's replicas attract few reads, the file is potentially cold. As creating a replica consumes network traffic and CPU usage, and the cost of mistakenly deleting a file replica is expensive, so we adopt a conservative way to determine if a file is cold. Only when both Equation (5.4) and Equation (5.5) are satisfied, file f_i is considered a cold file, represented by $C(f_i) = 1$.

After a file is labeled with its popularity (i.e., hot file or cold file), EAFR adjusts the number of its replicas according to its popularity. The details are presented in Section 5.2.1.3.

5.2.1.3 Adaptive File Replication

EAFR constantly monitors file popularity and adaptively tunes the number of replicas of a file based on whether it is hot or cold according to Section 5.2.1.2. If a file is a hot file and many of

its replica servers are overloaded, EAFR creates more replicas for this file to reduce overload degree and increase file availability. If a file is a cold file, EAFR reduces its replicas or transfers its replicas to standby servers. We first define r and ϵ ($\epsilon < r$), which are the minimum number of replicas a file needs to maintain in all servers and in hot servers, respectively, to guarantee file availability.

Consider a large cluster consisting of: 1) p hot servers, which are denoted by a set HS=(hs_1, hs_2,..., hs_p); 2) q cold servers CS=(cs_1, cs_2,..., cs_q); and 3) w standby servers SS=(ss_1, ss_2,..., ss_w). For a file f_i with r_i replicas, we use a set S_i=(s_1, s_2,..., s_{r_i}) to represent the servers that store its replicas. For server s_j, we define its service capacity (c_{c_j}) as the maximum number of concurrent file reads it can support. We use h_j to denote the concurrent reads s_j receives. If $h_j/c_{c_j} > \tau_c$, where τ_c is a threshold (e.g., 0.8), server s_j is considered as overloaded; otherwise, it is a lightly loaded server. The remaining capacity of a lightly loaded server s_j is calculated by $\phi_j = c_{c_j} - h_j$, which indicates the number of additional file requests it can handle.

At time T, if file f_i is hot ($H(f_i) = 1$), EAFR examines the load status of all server in S_i=(s_1, s_2,..., s_{r_i}). An extra replica is needed for f_i if more than γ_s ($0 < \gamma_s < 1$) fraction of these servers are overloaded, that is:

$$\sum_{s_j \in S_i} I(h_j/c_{c_j} > \tau_c) > r_i\gamma_s \ (0 < \gamma_s < 1). \tag{5.6}$$

When the inequality in Equation (5.6) is met, the current replica servers of f_i do not have enough capacity to handle an excessive number of file reads. Then, EAFR increases the number of replicas of f_i by 1. Otherwise, the current replica servers of f_i can handle the file reads even though f_i is hot, and there is no need to increase the number of replicas of f_i. The new replica will be placed in a hot server, so that it can serve new incoming file requests. The details of selecting the replica destination for the replica is presented in Section (5.2.2).

If $C(f_i) = 1$, f_i is cold and it draws few file reads. Then, the number of f_i's replicas can be reduced in order to save the storage. The rule of replica reduction is to delete a replica in a hot server, while still maintaining at least ϵ replicas in hot servers in order to guarantee file availability. In the replica reduction stage, EAFR first checks the number of replicas for f_i, i.e., r_i. If $r_i > r$ and the number of replicas in hot servers is larger than the threshold ϵ, EAFR chooses the server with the least remaining capacity and deletes the replica of f_i from it. That is, the selected server

s_j satisfies:

$$s_j = \arg\min_{s_j \in S_i} \{\phi_j\}. \tag{5.7}$$

In the case of $r_i = r$, if there are ϵ replicas in hot servers, no action is performed; if more than ϵ replicas are stored in hot servers, one replica is moved from a hot server to a cold server in order to save energy. EAFR selects a hot server s_j with the least remaining capacity according to Equation (5.7), and migrates the replica of f_i from s_j to a standby server. The standby server functions like hot server (i.e., it serves file requests) before turning to a cold server. Suppose the storage capacity of standby server s_i is c_{s_i}, if:

$$c_{s_i} - \sum_{j=1}^{m} b_j < \tau_s, \tag{5.8}$$

a standby server is ready to turn cold. b_j is the size of file j, m is the number of cold files that are currently stored in the standby server, and τ_s is the remaining storage threshold.

Algorithm 4: Pseudo-code of EAFR.

```
1:  Determine the popularity of file f_i
2:  if H(f_i) = 1:  //create one replica
3:      Select hs_j from the hot server pool; place replica in hs_i
4:  end if
5:  if C(f_i) = 1:  //reduce number of replica by one
6:      if number of replicas r_i > r
7:          Select s_j according to Equation (5.7)
8:          Delete the replica of f_i in hs_j
9:      else
10:         if more than ε replicas of f_i are stored in HS
11:             Migrate one replica of f_i from hs_i to ss_k
12:             if Equation (5.8) is satisfied for ss_k
13:                 ss_k turns into a cold server
14:             end if
15:         end if
16: end if
```

Algorithm 1 shows the pseudo-code of EAFR. This algorithm runs periodically to adaptively tune the number of replicas for each file. When a new replica of a file is created, hot servers that do not store the file's replica are candidates to be the new replica holders. In order to reduce the replication completion time and balance server load, EAFR selects the replica destination by considering both the expected transmission rate and server workload status.

5.2.2 Replica Destination Selection

When a network link suffers from congestion, the consequence is reflected in long write latency. In order to complete the file replication within a short time, the connection from source server to destination server should have high transmission rate. For this purpose, we can use an existing method [33] that monitors the network status (e.g., concurrent traffic, link utilization) and selects the links with light traffic [33]. However, such a monitoring method is complicated and requires additional monitoring overhead for large clusters. EAFR estimates a server's network condition based on recent completion time of transmitting a file to the server. This method is based on the rationale that the recent replication completion time can be an indicator of the server's network condition. To verify this rationale, we conducted an experiment as below.

We randomly selected a server as the source and 20 destination servers in Palmetto Cluster in Clemson University [40]. The source replicated 20 files to each destination server at the rate of once every 15 seconds. The file size is 100MB. We recorded the replication completion time on these servers, and showed the maximum, median and minimum replication completion time in Figure 5.4. We can make two observations from the figure. First, the replication completion time towards different servers exhibits obvious variance. The replications towards some servers (e.g., servers 1, 5 and 8) have smaller replication completion times than those towards other servers, while replications towards some servers (e.g., servers 2, 4 and 9) generally take longer completion time. This observation justifies the necessity and motivation of allocating the new replica to a server that has good network condition. Second, each server shows relatively stable replication completion time with a small variance between the maximum and minimum completion time. Thus, when multiple replicas are transmitted to the same server within a short period, the completion time for these replications should be similar. Therefore, a server's recent transmission speed can be used to predict the transmission speed in the near future. EAFR does not need to look into the link utilization and monitor the network congestion status when allocating a new file replica. It selects the replica destination based on the transmission speed of recent files.

To more accurately estimate the transmission delay of the next file based on the delays of previous file transmissions, we use an exponentially weighted moving average (EWMA) [77]. Using T as a time window size, EAFR calculates the average file transmission speed from source server s_s to destination serverss_d by sliding the time window. Then the transmission speed in the next time

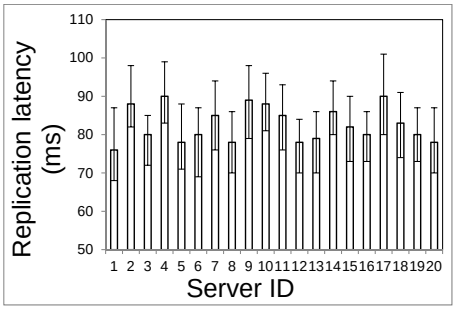


Figure 5.4: Replication completion time for different servers.

window (denoted by V_t) is calculated by:

$$V_t = \alpha \times Y_t + (1 - \alpha) \times V_{t-1} (0 < \alpha < 1), \tag{5.9}$$

where V_{t-1} is the estimated transmission speed in time window $t-1$, and Y_t represents the actual average transmission speed at time t. α is a constant used to control the degree of weighting decrease; a larger value of α discounts older observations faster. The weighting for each older EWMA data point decreases exponentially, but never reaches zero.

In addition to replica transmission latency, the replica destination must have enough storage capacity for new replicas. Also, as a new replica is created to serve an excessive amount of file requests, the replica should be placed in a server that has sufficient capacity to serve incoming file requests. Then, given a file from source server s_s, and a set of hot servers HS to place the new replica, EWMA selects a replica destination $s_d \in HS$ such that transmitting the file replica from s_s to s_d takes a short time and s_d has a high service capacity and enough storage capacity. For this purpose, EWMA first selects candidates from all hot servers HS that have enough storage space for this file replica.

EAFR calculates the expected transmission speed from s_s to all candidate servers, then orders the candidates based on the decreasing order of the transmission delays $ID_t = \{hs^1, hs^2, ...hs^m\}$. EAFR also orders the candidates based on the decreasing order of their remaining capacities $ID_r = \{hs^1, hs^2, ...hs^m\}$. A server having a faster transmission speed or a higher remaining capacity should have a higher probability to be selected. We use w_t and w_c to denote these two probabilities for a server. The

probability of the j^{th} server in these two ordered lists can be calculated by Equation 5.10.

$$\frac{1}{j} / \sum_{k=1}^{m} \frac{1}{k}. \tag{5.10}$$

The probability of selecting a server in the candidates is calculated by the weighted average of both of its w_t and w_c:

$$p = \beta \times w_t + (1 - \beta) \times w_c \tag{5.11}$$

We use vector $P = (p_1, p_2, ...p_m)$ to record all probabilities of selecting the candidate servers. Then, s_s selects the destination server based on P. The selection process first generates a random value x within the range of $[0, \sum_{k=1}^{m} p_k]$, then server with order y in the list P is selected by:

$$y = \begin{cases} 1 & \text{if } x < p_1 \\ j & \text{if } p_{j-1} \leq x < \sum_{k=1}^{j} p_k \text{ and } j > 1 \end{cases} \tag{5.12}$$

As we can see from Equation (5.12), the new replica is more likely to be allocated to a server with a high p value.

5.2.3 Dynamic Transmission Rate Adjustment

TCP incast occurs when a number of files from multiple storage servers are being sent to a server concurrently [78], and network congestion is likely to arise on the receiver side when multiple connections contend for bandwidth resources. TCP incast congestion increases the packet drop rate and reduces the transmission throughput, thus degrading network performance. To avoid incast congestion, while transmitting a file replica to a new server, EAFR dynamically adjusts transmission rate to prevent incast congestion.

Each server has a TCP receive window [69], which is a limited size of buffer that prevents a fast sender from overflowing a slow receiver's buffer. In EAFR, a destination server monitors its available bandwidth by using a bandwidth estimation tool [100] to detect sudden throughput burstiness. When it notices that a congestion is likely to occur, it reduces its receive window in proportion to the extent of congestion and notifies senders to reduce their transmission rates. If the destination node has enough available bandwidth to support larger transmission rates, it increases

the TCP receive window.

Algorithm 5: Pseudo-code of dynamic transmission rate adjustment.

Input: set of server S sending data through the link;
Output: adjust-down factor β_i and adjust-up factor δ_i for each sender in S;
for each $s_i \in S$ **do:**
  calculate available bandwidth on the link: $R_{c_b} = (c_b - u_b)/c_b$
  if $R_{c_b} < \eta_l$ //network capacity is highly utilized
    calculate adjust-down factor $\beta_i = \frac{i}{\sum_{k=1}^{p'} k} \times (\eta_1 - R_{c_b})$
  end if
  if $R_{c_b} > \eta_h$ //enough network capacity on the receiver side
    calculate adjust-up factor $\delta_i = \frac{i}{\sum_{k=1}^{p'} k} \times (R_{c_b} - \eta_h)$
  end if
  record β_i and δ_i for sender s_i
end for

Assume the link bandwidth capacity of the destination node is c_b (which is determined by its NIC and system settings), and the total bandwidth utilized by all incoming traffic is u_b. We then define the proportion of available bandwidth u_b on that link as:

$$R_{c_b} = (c_b - u_b)/c_b. \tag{5.13}$$

R_{c_b} is an indicator of potential oversubscribed bandwidth for a destination node. EAFR has two thresholds η_l and η_h to determine whether a network link capacity is highly utilized or underutilized. When $R_{c_b} < \eta_l$ (e.g., η_l=0.2), the network capacity is highly utilized and thus the receive window needs to be reduced. Suppose the destination server is receiving traffic from a number of connections from a set of servers $S = (s_1, s_2, ..., s_{p'})$. These connections have different priorities based on the connection establishment times. Since we aim to reduce the transmission latency of each flow, the connections with older establishing times have higher priorities. The senders are ordered in descending order of the priorities of their connections. For a sender with priority i, its adjust-down factor β_i is define as:

$$\beta_i = \frac{i}{\sum_{k=1}^{p'} k} \times (\eta_1 - R_{c_b}). \tag{5.14}$$

When $R_{c_b} > \eta_h$ (e.g., η_h=0.5), there is enough network capacity on the receiver side, then the receive window is increased for each connection to increase the transmission rate. The adjust-up factor for

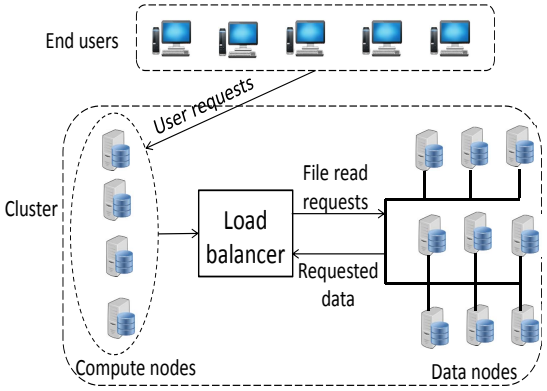


Figure 5.5: An overview of data flow in a cluster.

the server with priority i is δ_i defined by:

$$\delta_i = \frac{i}{\sum_{k=1}^{p'} k} \times (R_{c_b} - \eta_h). \tag{5.15}$$

After rate adjustment calculation, the destination node notifies the corresponding senders about the new transmission rates. Each sender then reduces its transmission rate by β_i times or increase its transmission rate by δ_i times.

Algorithm 5 shows the pseudo-code of dynamic transmission rate adjustment. For each sender s_i which establishes connection with the receiver, Algorithm 5 first calculates the link's available bandwidth capacity (Line 4); when the network capacity is highly loaded, EAFR reduces the sender's transmission rate by β_i; when the link's network capacity is lightly loaded, EAFR deliberately increases the sender's transmission rate by δ_i accordingly (Lines 5-7). The computation complexity of Algorithm 5 is $O(p)$, where p is the number of senders in the cluster. By dynamically adjusting the receive window in proportion to the extent of congestion, EAFR reduces the latency for file transmission in replications by avoiding incast congestions.

5.2.4 Network-aware Data Node Selection

Figure 5.5 shows an overview of data flow inside a cluster. As we can see, user requests are processed in compute nodes, and the compute nodes need to fetch files from data nodes where the requested files are stored. The file read latency is affected by two factors: 1) transmission delay between a compute node and a data node, and 2) queueing delay in a data node. As the intra-

rack connection has much higher bandwidth than the cross-rack connection, choosing a data node in the same rack as the requester computer node to transmit its requested file generates shorter latency than choosing a data node in a different rack. Also, when a cluster consists of heterogeneous servers, the service capacities of servers may vary significantly. A high-capacity data node can finish reading a file stored in its local disk faster than low-capacity nodes, resulting in smaller queue size and queueing delay. Accordingly, we propose a network-aware data node selection strategy by considering the aforementioned factors, i.e., a compute node should fetch its requested file from a data node within the same rack and with a short queue size.

Suppose compute node s_j needs to read file f_i when processing a user request, and f_i has r_i replicas stored in a number of servers. We use set S_i=(s_1, s_2,..., $s_{r_{i-1}}$) to represent all hot servers and standby servers that store f_i's replicas. We use h_k to denote the queue size of file read requests on server s_k, which is the number of reads that s_k has received but has not been processed. Algorithm 6 shows the pseudo-code of selecting data nodes for compute node s_j. For each file f_i that s_j needs to read from data nodes, Algorithm 6 first identifies all hot servers and standby servers storing f_i's replicas (Line 4). It then selects the data nodes within the same rack as s_j and puts them in a set S_i^1=(s_1, s_2,..., $s_{r_{i-1}^1}$) (Line 5). In order to reduce transmission delay between a compute node and a data node, we prefer intra-rack connection over cross-rack connection. Thus, if S_i^1 is not empty, Algorithm 6 selects data node s_k with the minimum queue size from S_i^1: $s_k = \text{argmin}_{s_k \in S_i^1}\{h_k\}$ (Lines 6-7); otherwise, it chooses a data node with the minimum queue size from S_i (Lines 8-10). The computation complexity of Algorithm 6 is $O(m \times \overline{r})$, where m is the number of files needed to read and $\overline{r}$ is the average number of replicas for these files.

Algorithm 6: Pseudo-code of network-aware data node selection for compute node s_j.

```
1: Input:  set of file F needed to fetch from data nodes;
2: Output: select a data node to read each file in F;
3: for each f_i ∈ F do:
4:     find servers storing f_i's replicas S_i=(s_1, s_2,..., s_{r_{i-1}})
5:     select S_i^1 from S_i that are in the same rack with s_j: S_i^1=(s_1, s_2,..., s_{r_{i-1}^1})
6:     if S_i^1 ≠ null   //find a data node within the same rack
7:         select a data node by s_k = argmin_{s_k ∈ S_i^1}{h_k}
8:     else   //find a data node within another rack
9:         select a data node by s_k = argmin_{s_k ∈ S_i}{h_k}
10:    end if
11:    record s_k as the data node to read file f_i
12: end for
```

5.2.5 Load-aware Replica Maintenance under Node Failure

As each server stores a large number of files, when a server failure happens, we create these lost files in other servers in order to maintain the minimum number of replicas per file. Suppose all files stored in a failed server are represented by F=(f_1, f_2,..., f_m). For each file f_i in F, we make a replica from a non-failed server (called source server) and place it in another non-failed server (called destination server). In order to minimize the energy consumption and time for the recovery process, we consider two objectives. First, we try to avoid waking up cold servers as it consumes extra energy and may lead to long recovery time. Second, we try to balance the incast traffic load caused by the file replications on the destination servers, i.e., the number of replicas allocated to destination servers, in order to avoid incast congestion in the destination servers and hence constrain the recovery latency.

Recall that each file has r_i replicas; at least ϵ replicas are stored in different hot servers, other replicas are stored in standby servers and cold servers. A hot server runs at active state and serves file requests; a standby server is a temporary hot server that stores cold files, and it turns to a cold server when its storage is fully utilized; and a cold server stores cold files, and it is in sleeping mode and does not serve file requests.

To achieve the first objective of avoiding rebooting the cold servers from the sleeping mode, we first try to select source servers from hot servers and standby servers. A cold server is waken up and selected as the source server only when a file stored in the failed server does not have any replicas stored in hot servers or standby servers. Specifically, for a file f_i in a failed server, we first put all hot servers and standby servers that store f_i in a server set S_i=(s_1, s_2,..., s_p). In order to find a server that has the maximum available service capacity to support the file reading operation, we first sort servers in S_i in decreasing order of their available service capacities. Then, we choose the first server as the source server. If there are no hot servers or standby servers storing file f_i (i.e., S_i is an empty set), we check the cold servers that store file f_i in the same manner and choose a cold server with the maximum available service capacity.

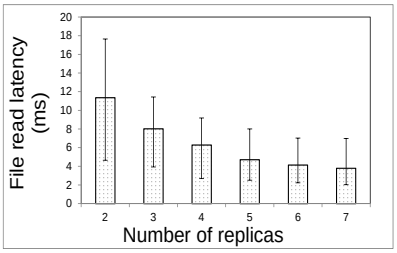
We use DS=(ds_1, ds_2,..., ds_w) to denote a set of candidate destination servers, DS is determined based on the popularity of f_i. If f_i is a hot file, DS is a set of non-failed hot servers; if f_i is a cold file, DS is a set of non-failed standby servers. Here, we do not choose a cold server as the destination server in order to avoid waking up the cold server for file replication, which otherwise

Algorithm 7: Pseudo-code of load-aware replica maintenance for a failed server

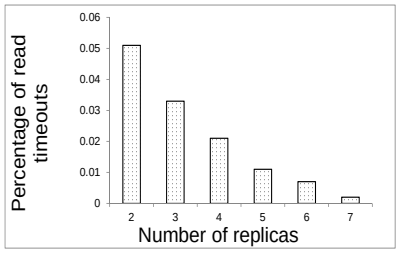
```
1: Input:  set of files F, counter = 1;
2: Output: source server and destination server for files in F;
3: for each f_i ∈ F do:
4:     order hot servers and standby servers storing f_i's replicas by their remaining service capacities,
   S_i=(s_1, s_2,..., s_p)
5:     select s_1 as the source server
6:     if S_i is empty
7:        choose a cold server with the maximum available service capacity as the source server
8:     end if
9:     while true   //select a destination server from DS
10:       choose server ds_k with index equaled to counter
11:       if c_{s_i} − Σ_{j=1}^{m} b_j ≥ τ_s,  //enough storage
12:           select ds_k as the destination server
13:           break
14:       else
15:           increase counter by 1
16:       end if
17:    end while
18:    record f_i's source server and destination server
19: end for
```

increases the recovery latency. To meet the second objective of balancing the incast traffic load of destination servers, we evenly place the replicas of all files of the failed server to DS by using a round robin [53] assignment method. We use a *counter* to record the index of the destination server candidates; *counter* increases from 1 to w in a circular manner, i.e., *counter* restarts from 1 after it reaches w. Assume $counter = k$ and we need to select a destination server for file f_i. We first examine the server ds_k whose index in DS equals the value of *counter*. If ds_k has enough storage capacity calculated by Equation (5.8), it is selected as the destination server for f_i where we store the replica of f_i; otherwise, we increase the value of *counter* by 1 until a server with enough storage capacity is detected. After we find a destination server for f_i, we add 1 to the *counter* and use it to identify the destination server for the next file f_{i+1}.

Algorithm 7 shows the pseudo-code of load-aware replica maintenance for a failed server, in which we choose a source server and a destination server for file f_i stored in the failed server. For each file f_i stored in the failed server, Algorithm 7 first orders all hot servers and standby servers storing f_i's replicas by their remaining service capacities (Line 4); it then selects a hot server or standby server with the maximum available service capacity as the source server (Line 5); if no hot servers or standby servers that store file f_i, it checks all cold servers that store file f_i and chooses a cold server with the maximum available service capacity as the source server (Lines 6-8); Algorithm 7 continues to select a destination server for file f_i by using a round robin assignment method (Lines

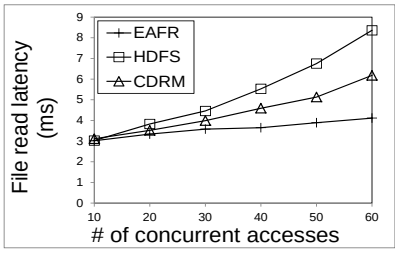


(a) File read response time.

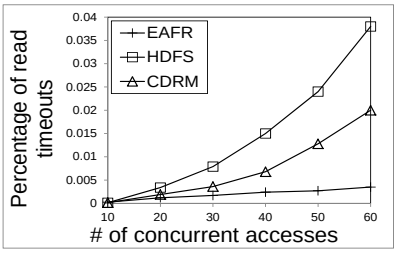


(b) Percentage of file read timeouts.

Figure 5.6: Performance under different number of replicas.



(a) File read response time.



(b) Percentage of file read timeouts.

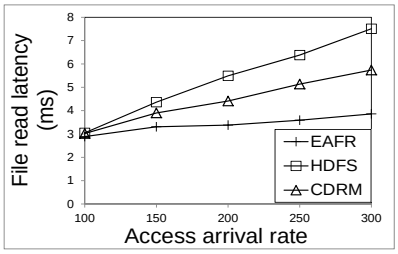
Figure 5.7: Performance under different # of concurrent accesses.

9-17). The computation complexity of Algorithm 7 is $O(m\overline{r}\log\overline{r})$, where m is the number of files in the failed server and $\overline{r}$ is the average number of replicas for these files.

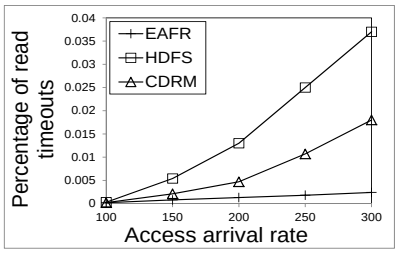
5.3 Performance Evaluation

We conducted trace-driven experiments in a large-scale HPC cluster located in Clemson University's Palmetto Cluster [40] which has 1,978 nodes. We deployed EAFR on 300 servers evenly scattered in 10 racks. The storage capacities of these servers were randomly chosen from (250GB, 500GB, 750GB) [103]. We compared EAFR with HDFS [98] and CDRM [109] that are similar to our work. In HDFS, every file has a fixed number of 3 replicas placed across different randomly selected servers. CDRM aims to deal with server failures. In our experiment, CDRM creates 2 replicas for each file initially, it increases the number of replicas to maintain the required file availability 0.98 for server failure probability 0.1, and required file availability 0.8 for server failure probability 0.2. CDRM allocates the newly created replica to the server with the least concurrent reads.

Unless otherwise specified, the distributions of file reads and writes follow those in the CTH trace data [105] and each file read request was forwarded to a randomly selected server that owns the replica of the requested file. This trace records 3,972,284 I/O calls on 16,566 files during about



(a) File read response time.



(b) Percentage of file read timeouts.

Figure 5.8: Performance under different access arrival rates.

4 hours in a large cluster with 3300 client size. We created 50,000 files and randomly placed them on the servers. The sizes of 16,566 files were set according to the CTH trace data, and the sizes of other files were chosen from the range (100KB, 10GB). The server capacity follows the normal distribution [98] with a mean of 15 and variance of 10. When the number of concurrent file reads is larger than a server's service capacity, new coming file requests will be put into a waiting queue until the server has available capacity. The remaining storage threshold ∇ was set to 10GB; other system parameters were set as: τ_u=20, τ_l=10, σ_u=8, σ_l=1, $r = 2$ and ϵ=1. The power consumption for different types of servers was calculated based on Table 5.2. We randomly selected 70% of servers as hot servers, and 30% of servers function as standby servers. A standby server with full storage turns into a cold server. The experiment runs 2 days by repeatedly using the read rates from the trace data. We are interested in the following performance metrics:

- *File read latency*: the latency from a user requests a file until the user receives a response from the server.
- *Replication latency*: the latency from when a file replication is initiated until the replication operation is finished.
- *Energy cost*: the server power consumption in kilowatt hour (kWh) in each day.
- *Load balance status* including: 1) *server utilization*, which is defined as the ratio of the number of concurrent file requests a server is serving over the server's capacity; and 2) *percentage of overloaded servers* that are defined as the servers with more than 80% utilization.
- *Memory consumption*: the storage usage to store all file replicas (including the original copy) in the system.

- *Maintenance overhead.* An update's maintenance overhead is defined as the product of the latency of this update and the update message size. A file's maintenance overhead is the sum of the maintenance overheads of the updates on all of its replicas.

- *Recovery latency*: the time span from when the creation of file replicas in a failed server is initiated until all file replicas stored in the failed servers are recovered.

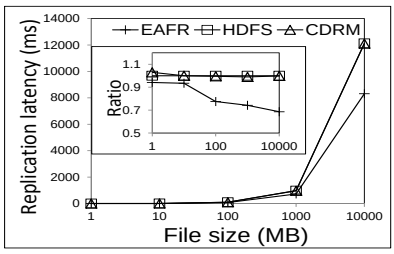
5.3.1 Experimental Results for Overall Performance

5.3.1.1 File Read Response Latency

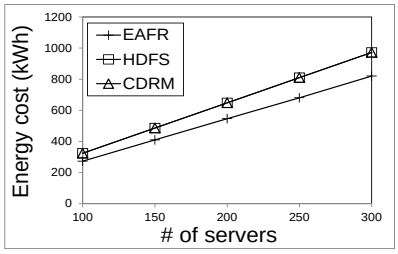
When a hot file attracts a large amount of concurrent reads, some file requests may contend for server capacity and network bandwidth, and hence suffer from response latency.

Number of replicas. We first study the effectiveness of creating extra file replicas for hot files in reducing the file read response time. We selected 20 random files, varied the number of replicas for each file, and generated 60 concurrent read requests towards each file. Figure 5.6(a) shows the 1st percentile, median and 99th percentile read response time when each file has a different number of replicas. We see that more replicas lead to decreased read response time, i.e., when the number of replicas for each file increases from 2 to 7, the median read response time drops from about 11ms to 4ms. This is due to the reason that when there are only 3 replicas allocated in 3 individual servers, large numbers of concurrent read requests are flooded to the same server, and some read requests need to wait if the server capacity is already fully occupied by requests. However, when more replicas are created in different servers, more server capacity can be utilized to serve the read requests. Thus, concurrent read requests are forwarded to different servers and are less likely to contend for server capacity. We define 10ms as the required latency threshold, and record the percentage of file read requests that are served past the required latency. Figure 5.6(b) shows the percentage of file read timeouts when a different number of replicas are created for each file. We see that the percentage of read timeouts drops gradually when more replicas are created for each file for the same reason as in Figure 5.6(a). That is to say, creating more replicas for hot files can prevent resource contention between excessive number of synchronous requests. Figure 5.6(a) and Figure 5.6(b) prove the rationale of *EAFR* that increasing the replicas of hot files can shorten the read response time and increase data availability.

Number of concurrent read requests. We then varied the number of concurrent read

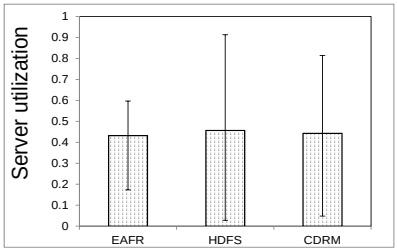


(a) Replication latency for files of various sizes.

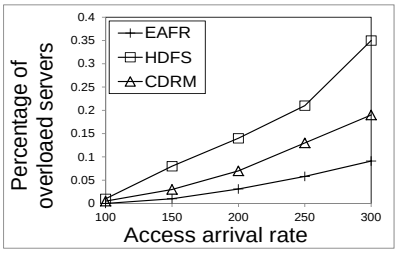


(b) Energy consumption per day.

Figure 5.9: Replication latency and energy consumption.



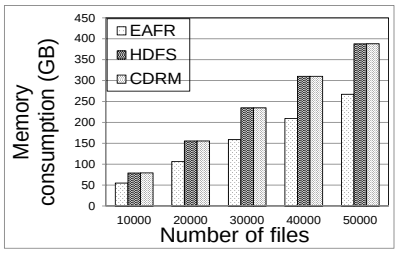
(a) Server utilization.



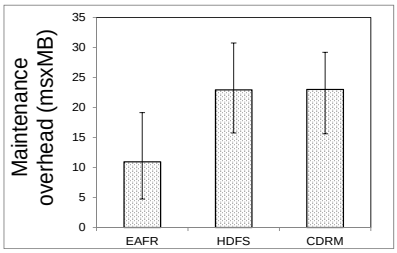
(b) Percentage of overloaded servers.

Figure 5.10: Load balance status.

requests by replacing one read in the trace data by x reads. x is varied from 10 to 60 increasing by 10 in each step. Figure 5.7(a) shows the average file read response time with different number of concurrent reads to the same file (i.e., x) in the system. We see that the response latency increases as the number of concurrent reads increases. This is because servers can serve a limited number of requests at a time and new file requests must wait in queues until the servers have available capacity. We also see that *CDRM* yields less latency than *HDFS*. This is because *CDRM* chooses the server with the least workload as the replica destination, then the server storing the new file replica is likely to have enough capacity to serve file requests. *HDFS* randomly selects replica destination, which may not have enough capacity to handle requests. Thus *HDFS* incurs longer latency than other two methods as read requests are likely to wait for server response. *EAFR* produces the least read latency because it adaptively increases the number of replicas for hot files, and the new replicas share the read workload of hot files. Thus, a large number of concurrent file requests are not likely to overload the servers and wait for response. As *CDRM* does not consider file popularity in replication, file requests towards hot files still need to contend for server capacity.Figure 5.7(b) shows the percentage of file read timeouts versus the number of concurrent reads. We see that the result follows *HDFS*>*CDRM*>*EAFR* for the same reasons as in Figure 5.7(a).

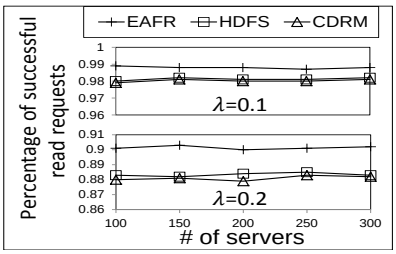


(a) Memory consumption.

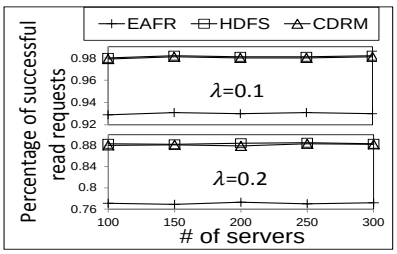


(b) Maintenance overhead.

Figure 5.11: Overhead.



(a) File availability when r=3.



(b) File availability when r=2.

Figure 5.12: File availability.

Access arrival rate. Access arrival rate is defined as the number of file requests generated in the system in each second. In order to investigate the performance of *EAFR* under different workload distributions, we varied the file read arrival rates by changing the time interval between two consecutive reads in the trace data (e.g., reduce the time interval between two successive reads to increase the file read rate). Figure 5.8(a) shows the average file read response latency with different arrival rates. We can see that as the access arrival rate grows from 100 to 300 reads per second, *HDFS* and *CDRM* both rise quickly. This is due to the reason that a limited number of replicas are insufficient to serve large number of read requests, and as the access arrival rate gets higher, more requests are likely to stay in waiting queue. *EAFR* adaptively increases the number of replicas for hot files, thus produces less file read response latency than *HDFS* and *CDRM* due to same reasons as in Figure 5.7(a). Figure 5.8(b) shows the percentage of read timeouts with different arrival rates. We see that *EAFR* produces fewer read timeouts than *HDFS* and *CDRM* for the same reasons as in Figure 5.7(b). As applications (such as some web-based applications) deployed on large clusters need to provide prompt service to their clients, and the above figures show the effectiveness of *EAFR* in reducing file read latency and providing high quality support for time-sensitive applications.

5.3.1.2 Replication Completion Time

We grouped the files with the same size (ranging from 1MB to 10,000MB) together and calculated the average replication latency of each group of files. Figure 5.9(a) shows the replication completion time for different file groups. We also set the replication completion time of *HDFS* as base and plot the ratio of other methods' replication completion time over the base in the embedded figure. We see that replication operations can be completed with short latency for small files due to the high-speed network connections on Palmetto clusters. However, the replication completion time grows rapidly for files with large sizes. *EAFR* speeds up the file replication especially for large files, and the improvement reaches about 30% when the file size is 10,000MB. This is because *EAFR* predicts the transmission speed based on previous file transmission experience and selects the server with a high transmission rate with high probability, i.e., file replicas are more likely to be allocated to servers with good network condition. Also, it dynamically adjusts the file transmission rate during replication process in order to prevent incast congestion on the receiver side, thus reducing transmission latency.

5.3.1.3 Energy Efficiency

We examined the effectiveness of *EAFR* in reducing energy consumption. We set the power consumption of different genre of servers according to Table 5.2. Figure 5.9(b) shows the total amount of energy consumption per day for different methods when various number of servers are used in the cluster. Due to the adoption of cold servers to store cold files that are rarely read by clients, *EAFR* manages to reduce the power consumption by more than 150kWh per day in a cluster consisting of 300 servers. Given a fixed number of servers in the cluster, *EAFR* aims to allocate popular files to servers that are guaranteed to be on (hot servers), and it stores some replicas of cold files in cold servers (in sleeping mode), which results in substantial power saving. It is worth noting that while the adoption of cold servers can reduce the energy consumption in large cluster, performance of the cluster in serving file requests is not compromised, which is demonstrated in the previous figures.

5.3.1.4 Load Balance Status

It is crucial to constrain the workloads of servers under their capacities (i.e., achieving load balance), which help reduce file read response latency. Server utilization is an indicator of how balance the file requests are distributed among servers in the system. For each server, we sampled 10 utilization values within 10 minutes at a frequency of once per minute, then selected the highest value as the server's utilization to report. Figure 5.10(a) plots the 1st percentile, median and 99th percentile of server utilization of different methods. We see that *EAFR* achieves better load balance than *CDRM* and *HDFS* with a smaller 99th percentile value and a larger 1st percentile value. *EAFR* adaptively increases the number of replicas for hot files to serve excessive file requests and reduces the number of replicas for cold files. Also, it creates new replicas in servers with the highest remaining capacity with a high probability. As the workloads are better balanced in *EAFR*, it can effectively prevent the servers storing hot files from becoming overloaded, and file requests are less likely to be blocked. We then tested the performance of *EAFR* under different workload distributions by varying the file read arrival rates using the same method as in Section 5.3.1.1. Figure 5.10(b) shows the percentage of overloaded servers during the experiment in the system. We see that the percentage of overloaded servers rises gradually with increased read arrival rates for all methods, as more server capacity is consumed to serve read requests. *EAFR* maintains the least percentage of overloaded servers due to the same reason as in Figure 5.10(a).

5.3.1.5 Overhead

Figure 5.11(a) shows the memory consumption of different methods when a various number of original files are stored in the system. We see that *EAFR* has lower memory consumption than other two methods because cold files only maintain 2 replicas in the system, and small amount of extra replicas are created for hot files to meet the short-term intensive read requests. In *HDFS*, keeping a fixed number of 3 replicas consumes more storage resource than *EAFR*. *CDRM* maintains 2 replicas for each file initially, and increases the number of replicas to meet the required file availability, so it demands more memory consumption than *EAFR*.

When a file is modified, each replica of the file should be updated in order to maintain consistency, and the update of file is accomplished by performing a write operation to synchronize each of its replica. In *EAFR*, as cold servers do not serve file read requests, when a file is updated,

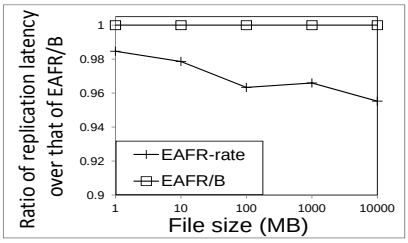
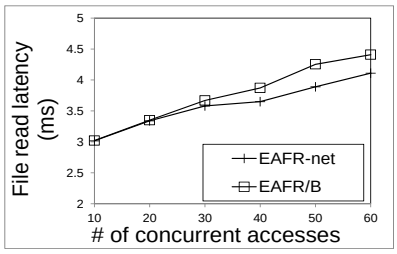
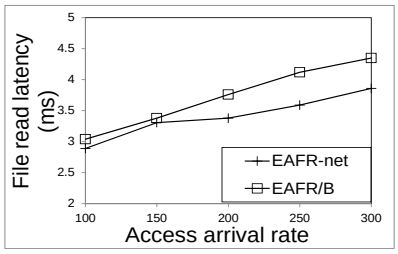


Figure 5.13: Replication latency for files of various sizes.



(a) Performance under different # of concurrent accesses.



(b) Performance under different access arrival rates.

Figure 5.14: File read response time.

the writes need not be sent to its replicas stored in cold servers immediately. Instead, the updates of replicas in cold servers are postponed, until the servers are switched from sleeping mode to active mode. More precisely, the cold servers are woken up once per week to check for file updates. Whenever a file replica is dirty (i.e., not updated), a write operation is performed to synchronize the replica. We generated the updates of files from the trace data, and defined a file's maintenance overhead as the product of total amount of latency (in ms) to send writes to all its replicas multiplied by the size of the file (in MB). Figure 5.11(b) shows the 1st percentile, median and 99th percentile of maintenance overhead for all methods. We see *EAFR* displays substantially smaller median, 1st percentile and 99th percentile maintenance overhead than the other two methods due to three reasons. First of all, *EAFR* creates a smaller number of replicas for cold files compared to *CDRM* and *HDFS*, thus, fewer writes are needed if a cold file needs to be updated. Secondly, the replicas in cold servers in *EAFR* do not need updates when the servers are in sleeping mode. Thirdly, *EAFR* tries to reduce network congestions in file replication, which may also help reduce the updating latency. As a result, *EAFR* produces relatively low maintenance overhead.

5.3.1.6 Server Failure Resilience

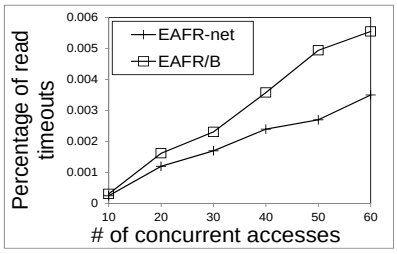
We tested *EAFR*'s resilience to server failures though it is not *EAFR*'s objective. Each server has a failure probability λ, and when all servers storing a file's replicas fail, requests for this file fail. We measured the file availability as the percentage of available files among all files stored in the system, a good file system in cluster should provide high file availability to clients. Figure 5.12(a) shows the percentage of successful read requests when $\lambda = 0.2$ and $\lambda = 0.1$, and the minimum number of replicas in *EAFR* is 3. We see that *EAFR* achieves the highest percentage of successful file requests. This is because *EAFR* creates extra replicas for hot files, which in turn increase the percentage of successful requests of hot files in server failures. *HDFS* keeps a fixed number of 3 replicas for each file and achieves lower percentage of successful requests than *EAFR*. *CDRM* stops increasing the file replicas when the percentage is higher than 0.8. Figure 5.12(b) shows the percentage of successful read requests when the minimum number of replicas in *EAFR* is 2. We see that *EAFR* provides relatively lower percentage of successful read requests than the other two methods due to the reason that only 2 replicas are maintained for most files. *CDRM* increases the number of file replicas to maintain a required file availability, so it provides high file availability under different server failure probabilities.

5.3.2 Experimental Results for Enhancement Strategies

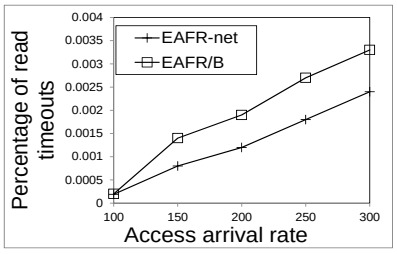
In the following, we show the effectiveness of each of our proposed strategies for enhancement: i) dynamic transmission rate adjustment strategy, ii) network-aware data node selection strategy and iii) load-aware replica maintenance strategy. In the following figures, we use *EAFR/B* to denote the basic EAFR without any enhancement strategies.

5.3.2.1 Effectiveness of Dynamic Transmission Rate Adjustment Strategy

Figure 5.13 shows the replication completion time for different file groups with and without the dynamic transmission rate adjustment strategy (denoted by *EAFR-rate* and *EAFR/B*). We set the replication completion time of *EAFR/B* as base and plot the ratio of *EAFR-rate* and *EAFR/B*'s replication completion time over the base. We see that *EAFR-rate* effectively reduces replication time compared to *EAFR/B*. The reason is that *EAFR-rate* dynamically adjusts the senders' transmission rates based on the receiver's bandwidth consumption, which can prevent incast congestion on the



(a) Performance under different # of concurrent accesses.



(b) Performance under different access arrival rates.

Figure 5.15: Percentage of file read timeouts.

receiver side. As a result, the receiver is not likely to be congested and file replication operations can be completed within short latency.

5.3.2.2 Effectiveness of Network-aware Data Node Selection Strategy

We denote EAFR with and without applying the proposed network-aware data node selection strategy by *EAFR-net* and *EAFR/B*, respectively. In *EAFR/B*, a compute node fetches its requested file from a randomly selected data node among the data nodes storing the file's replicas.

Figure 5.14(a) and Figure 5.14(b) show the average file read response time with different number of concurrent reads and different access arrival rates, respectively. We see that the response latency increases as the number of concurrent reads increases due to the same reason as in Figure 5.7(a). We also see that *EAFR-net* reduces the file read response latency. A compute node in *EAFR-net* tends to fetch files from data nodes within the same rack as the requester computer nodes to minimize the file transmission time, and from data nodes with small queue sizes to reduce the queueing delay. Therefore, a compute node can finish reading a file within shorter latency in *EAFR-net* than that in *EAFR/B*. We also notice that the reduction in response latency becomes larger when there are a larger number of concurrent reads and access arrival rates in the system. This is because when the number of concurrent reads and access arrival rates increase, servers in *EAFR/B* are more likely to be overloaded as the read requests are assigned to servers without considering their queue sizes, which leads to file read response time increase. On the other hand, *EAFR-net* aims to assign read requests to servers with small queue sizes, which reduces the file read latency compared to *EAFR/B*. Figure 5.15(a) and Figure 5.15(b) show the percentage of file read timeouts with different number of concurrent reads and different access arrival rates, respectively. We see that *EAFR-net* reduces the percentage of file read timeouts due to the same reason as explained in

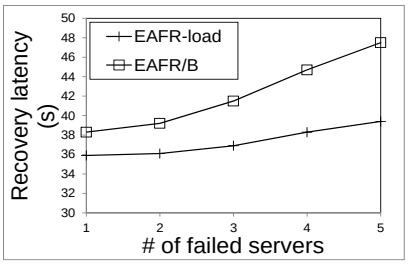


Figure 5.16: Server maintenance latency for various number of failed servers.

Figure 5.14(a) and Figure 5.14(b). *EAFR-net* aims to minimize file read response time by letting a compute node to fetch files from data nodes within the same rack and from data nodes with small queue sizes; while *EAFR/B* randomly assigns read requests to data nodes, which results in high percentage of file read timeouts. Figure 5.14(a), Figure 5.14(b), Figure 5.15(a) and Figure 5.15(b) show the effectiveness of our proposed network-aware data node selection strategy in reducing file read response time.

5.3.2.3 Effectiveness of Load-aware Replica Maintenance Strategy

We denote EAFR with and without applying the proposed load-aware replica maintenance strategy by *EAFR-load* and *EAFR/B*, respectively. In this experiment, we randomly selected a number of servers as failed servers every 30 minutes and recovered all file replicas stored in each failed server. We then recorded the average recovery latency. *EAFR/B* randomly selects a source server for a file's replica without considering server capacities, and also randomly selects a destination server with enough storage capacity to place a file's replica without balancing the number of replicas stored in each destination server to constrain the incast network load. Figure 5.16 shows the average replica recovery latency for a various number of failed servers. We see that as the number of failed servers increases, both *EAFR-load* and *EAFR/B* generate longer replica recovery latency. The reason lies in that more failed servers lead to the replication of a larger number of file replicas. As more files are transmitted from source servers to the identified destination servers, these transmissions need to compete for limited bandwidth capacity and thus lead to longer transmission delay. We also see that *EAFR-load* improves *EAFR/B* by generating shorter recovery latency. In *EAFR-load*, we aim to select servers with the maximum remaining service capacity as source servers, so file can be read from source servers with short latency. Also, compared to *EAFR/B*, *EAFR-load* can balance the load (i.e., the number of replicas allocated) of destination servers as it aims to evenly allocate file

replicas to all destination servers, which effectively prevent incast congestion in destination servers and generate shorter file transmission time. As a result, *EAFR-load* generates shorter recovery latency.

Chapter 6

Conclusion

A high-performance cloud can reduce the cost of both cloud providers and customers, while providing high application performance to cloud clients. To build a high-performance cloud, in this dissertation, we propose three methods to solve the challenges in delivering and storing contents on the cloud. Specifically, the three methods aim to provide a cost-efficient gaming system to support thin-client MMOG, an inter-datacenter video scheduling algorithm for video transmission on the cloud, and an adaptive file replication algorithm for cloud storage system. We summarize the works in this dissertation below.

Firstly, cloud gaming is a very promising model for thin-client MMOG since it frees players from this requirement, but it faces formidable challenges that prevents it from achieving high user QoE and low cost. We propose *CloudFog*, which leverages supernodes functioning as "fog" to connect the cloud to users. The cloud conducts the intensive computation for producing game state and sends update information to supernodes. The supernodes then generate game videos to stream to players. To select a suitable supernode that can provide satisfactory game video streaming service, we propose a reputation based supernode selection strategy. Considering that different games have different degrees of response latency tolerance and packet loss tolerance, we propose a receiver-driven encoding rate adaption strategy to balance these two factors in guaranteeing QoS. Since social friends in online games tend to play game together, we assign these players to the same server in a datacenter to reduce the interactions of servers to further reduce the latency. We also propose a dynamic supernode provisioning strategy to deal with user churns and relieve server loads. As a result, CloudFog reduces response latency and bandwidth consumption and increases user coverage.

These advantages are verified by our experiments on the PeerSim simulator and the PlanetLab real-world testbed.

Secondly, to provide video streaming services to users across different regions, cloud providers need to transfer video contents between different datacenters. Such inter-datacenter transfers are charged by ISPs under the percentile-based charging models. We take advantage of the particular characteristic of these models and propose EcoFlow to minimize cloud providers' payment costs on inter-datacenter traffics. EcoFlow is an economical and deadline-driven video transfer strategy. It first estimates the total volume of video traffic needed to be transmitted between every two datacenters within a time period, compares it with the charging volume and calculates the under-utilized traffic volume on each link. EcoFlow then schedules video flows with the objective that these flows do not incur additional charges on the link and guaranteing that each video flow meets its transmission deadline. Finally, the under-utilized links with low traffic burden are used to build alternating paths for video flows that are estimated to miss their deadlines. To enhance EcoFlow, we also propose setting each link's initial charging volume to reduce the scheduling latency at the beginning of the charging period. We further discuss how to deal with link available bandwidth prediction errors and lack of prior knowledge of the charing volume. Moreover, we design the implementation of EcoFlow in both a centralized manner and a distributed manner. Experimental results on PlanetLab and EC2 show the effectiveness of EcoFlow in reducing bandwidth costs and at the same time guaranteing that each video flow meets its transmission deadline for inter-datacenter video transfers.

Finally, in this dissertation, we propose EAFR to reduce file read latency, power consumption and replication completion latency. EAFR adaptively increases the number of replicas for hot files to alleviate intensive file request loads, and thus reduce the file read latency, and also decreases the number of replicas for cold files without compromising their read efficiency. Some replicas of cold files with few accesses are transferred to cold servers with 0% CPU utilization to save power. EAFR selects servers with sufficient capacities to place new replicas to shorten replication completion time and avoid overloading servers. EAFR also has a transmission rate adaptation strategy to further prevent potential incast congestion, a network-aware data node selection strategy to reduce file read latency and a load-aware replica maintenance strategy to maintain a certain number of replicas upon server failures. Experimental results from a real-world large cluster show the effectiveness of EAFR and the proposed strategies in meeting the demands of file systems in large clusters.

The future work will be three folds. First, in cost-efficient gaming system to support thin-

client MMOG, we will study the security issues such as dealing with malicious supernodes and preventing cheating behaviors in *CloudFog*, as well as study how to evaluate the user Quality of Experience (QoE) when using the CloudFog system; Second, for inter-datacenter video scheduling, we will investigate how to improve the routing algorithm of finding alternating paths for video flows. Third, for adaptive file replication in cloud storage system, we will study increasing data locality in replica placement, and determining the optimal number of cold servers to maximize energy saving without compromising the file read efficiency.

Bibliography

[1] How to batch render in Sony Vegas. http://sony-vegas.wonderhowto.com/how-to/batch-render-sony-vegas-329667/, [Accessed in Dec, 2015].

[2] Latency (lag) vs win rate in League of Legends. https://www.reddit.com/r/dataisbeautiful/comments/1t23a0/latency_lag_vs_win_rate_in_league_of_legends_oc/, [Accessed in J, 2016].

[3] The difference between upload and download speed for broadband DSL. http://fetchsoftworks.com/fetch/help/Contents/Tu torial/SlowUploads.html, [Accessed in June, 2016].

[4] S. Raza A. Nazir and C. Chuah. Unveiling dacebook: a measurement study of social network based applications. In *Proc. of SIGCOMM*, 2008.

[5] L. Abad and Y. Luand R. Campbell. Dare: Adaptive data replication for efficient cluster scheduling. In *Proc. of CLUSTER*, 2011.

[6] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z. Zhang. Unreeling netflix: Understanding and improving multi-cdn movie delivery. In *Proc. of INFOCOM*, 2012.

[7] U.S. Energy Information Administration. http://www.eia.gov, [accessed in feb 2015].

[8] Service Level Agreements. http://azure.microsoft.com/en-us/support/legal/sla/, [Accessed in Feb 2015].

[9] S. Ahmad, C. Bouras, E. Buyukkaya, R. Hamzaoui, A. Papazois, A. Shani, G. Simon, and F. Zhou. Peer-to-peer live streaming for Massively Multiplayer Online Games. In *Proc. of P2P*, 2012.

[10] D. Ahmed and S. Shirmohammadi. A microcell oriented load balancing model for collaborative virtual environments. In *Proc. of VECIMS*, 2008.

[11] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proc. of NSDI*, 2010.

[12] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proc. of NSDI*, 2012.

[13] H. Amur, J. Cipar, V. Gupta, G. Ganger, M. Kozuch, and K. Schwan. Robust and flexible power-proportional storage. In *Proc. of SoCC*, 2010.

[14] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proc. of EuroSys*, 2011.

[15] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[16] A. Ashraf, F. Jokhio, T. Deneke, S. Lafond, I. Porres, and J. Lilius. Stream-based admission control and scheduling for video transcoding in cloud computing. In *Proc. of CCGrid*, 2013.

[17] N. Bansal and M. Harchol-Balter. Analysis of srpt scheduling: investigating unfairness. In *Proc. of SIGMETRICS/Performance*, 2001.

[18] L. Barroso and U. Holzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. 2009.

[19] A. Beloglazov and R. Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *CCPE*, 24(13):1397–1420, 2012.

[20] C. Bezerra, J. Comba, and C. Geyer. Adaptive load-balancing for MMOG servers using KD-trees. *Computers in Entertainment*, 10(3):5, 2012.

[21] C. Bezerra and C. Geyer. A load balancing scheme for massively multiplayer online games. *Multimedia Tools Appl*, 2009.

[22] N. Bila, E. de Lara, K. Joshi, A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan. Jettison: efficient idle desktop consolidation with partial vm migration. In *Proc. of EuroSys*, 2012.

[23] J. Blackburn, R. Simha, N. Kourtellis, X. Zuo, M. Ripeanu, J. Skvoretz, and A. Iamnitchi. Branded with a scarlet "C": cheaters in a gaming social network. In *Proc. of WWW*, 2012.

[24] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *Proc. of SIGCOMM*, 2012.

[25] B. Van De Bovenkamp, S. Shen, A. Iosup, and F. Kuipers. Understanding and recommending play relationships in online social gaming. In *Proc. of COMSNETS*, 2013.

[26] E. Carlini, M. Coppola, and L. Ricci. Integration of P2P and Clouds to support Massively Multiuser Virtual Environments. In *Proc. of NetGames*, 2010.

[27] J. Chase and R. Doyle. Balance of power: Energy management for server clusters. In *Proc. of HotOS*, 2001.

[28] J. Chen, M. Arumaithurai, X. Fu, and K. Ramakrishnan. Gaming over COPS: A Content Centric Communication Infrastructure for Gaming Applications. In *Proc. of ICDCS*, 2012.

[29] K. Chen, P. Huang, and C. Lei. Game Traffic Analysis: An MMORPG Perspective. *Computer Networks*, 50(16):3002–3023, 2006.

[30] Y. Chen, A. Ganapathi, A. Fox, R. Katz, and DPatterson. Statistical workloads for energy efficient mapreduce. In *Technical report, UC, Berkeley*, 2010.

[31] Xu Cheng and Jiangchuan Liu. Nettube: Exploring social networks for peer-to-peer short video sharing. In *Proc. of INFOCOM*, 2009.

[32] Z. Cheng, Z. Luan, Y. Meng, Y. Xu, D. Qian, A. Roy, N. Zhang, and G. Guan. Erms: An elastic replication management system for hdfs. In *Proc. of CLUSTER Workshops*, 2012.

[33] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proc. of SIGCOMM*, 2013.

[34] M. Chowdhury, M. Zaharia, J. Ma, M. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *Proc. of SIGCOMM*, 2011.

[35] S. Choy, B. Wong, G. Simon, and C. Rosenberg. EdgeCloud: A New Hybrid Platform for On-Demand Gaming. Technical Report CS-2012-19, University of Waterloo, 2012.

[36] S. Choy, B. Wong, G. Simon, and C. Rosenberg. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *Proc. of NetGames*, 2012.

[37] M. Claypool. Motion and scene complexity for streaming video games. In *Proc. of FDG*, 2009.

[38] Amazon Elastic Compute Cloud.
http://aws.amazon.com/ec2/, [Accessed in June. 2016].

[39] Amazon Elastic Compute Cloud.
http://aws.amazon.com/ec2/, [Accessed in June. 2016].

[40] Palmetto Cluster. http://citi.clemson.edu/palmetto/index.html, [Accessed in Feb 2015].

[41] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of www client-based traces. Technical report 1995-010, boston univ., 1995.

[42] A. Dhamdhere and C. Dovrolis. Isp and egress path selection for multihomed networks. In *Proc. of INFOCOM*, 2006.

[43] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[44] J. Douceur, J. Mickens, T. Moscibroda, and D. Panigrahi. Collaborative measurements of upload speeds in p2p systems. In *Proc. of INFOCOM*, 2010.

[45] Y. Feng, B. Li, and B. Li. Jetway: Minimizing costs on inter-datacenter video traffic. In *Proc. of Multimedia*, 2012.

[46] P. Gao, A. Curtis, B. Wong, and S. Keshav. It's not easy being green. *ACM SIGCOMM Computer Communication Review*, 42(4):211–222, 2012.

[47] I. Goiri, J. Guitart, and J. Torres. Economic model of a Cloud provider operating in a federated Cloud. *Information Systems Frontiers*, 14(4):827–843, 2012.

[48] D. Goldenberg, L. Qiu, H. Xie, Y. Yang, and Y. Zhang. Optimizing cost and performance for multihoming. In *Proc. of SIGCOMM*, 2004.

[49] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. *Proc.of SIGCOMM*, 2009.

[50] A. Greenberg, J. Hamilton, D. Maltz, and P. Patel. The Cost of a Cloud: Research Problems in Data Center Networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, 2009.

[51] C. Hellstrom, K. Nilsson, J. Leppert, and C. Aslund. Influences of motives to play and time spent gaming on the negative consequences of adolescent online computer gaming. *Computers in Human Behavior*, 28(4):1379–1387, 2012.

[52] M. Hemmati, A. Javadtalab, A. Shirehjini, S. Shimohammadi, and T. Arici. Game as Video: Bit Rate Reduction through Adaptive Object Encoding. In *Proc. of NOSSDAV*, 2013.

[53] R. Hendrickson. *The Facts on File encyclopedia of word and phrase origins*. Facts on File, 1997.

[54] T. Hobfeld, R. Schatz, M. Varela, and C. Timmerer. Challenges of QoE management for cloud applications. *IEEE Communications Magazine*, 50(4):28–36, 2012.

[55] K. Hoffman, D. Zage, and C. Nita-Rotaru. A survey of attack and defense techniques for reputation systems. *CSUR*, 42(1):1, 2009.

[56] C. Huang, C. Hsu, Y. Chang, and K. Chen. GamingAnywhere: An Open Cloud Gaming System. In *Proc. of MMSys*, 2013.

[57] C. Huang, J. Li, and K. W. Ross. Can internet video-on-demand be profitable? In *Proc. of SIGCOMM*, 2007.

[58] Y. Huang, T. Fu, D. Chiu, J. Lui, and C. Huang. Challenges, design and analysis of a large-scale p2p-vod system. In *Proc. of SIGCOMM*, 2008.

[59] A. Hussam, P. Lonnie, and W. Hakim. Racs: A case for cloud storage diversity. In *Proc. of SoCC*, 2010.

[60] S. Ibrahim, B. He, and H. Jin. Towards pay-as-you-consume cloud computing. In *Proc. of SCC*, 2011.

[61] Gaikai. Inc. http://www.gaikai.com/, [accessed in feb 2015].

[62] Onlive. Inc. http://www.onlive.com/, [Accessed in Feb 2015].

[63] M. Jarschel, D. Schlosser, S. Scheuring, and T. Hobfeld. An Evaluation of QoE in Cloud Gaming Based on Subjective Tests. In *Proc. of IMIS*, 2011.

[64] M. Jarschel, D. Schlosser, S. Scheuring, and T. Hobfeld. Gaming in the clouds: QoE and the users' perspective. *Mathematical and Computer Modelling*, 2011.

[65] R. Kaushik, T. Abdelzaher, R. Egashira, and K. Nahrstedt. Predictive data and energy management in greenhdfs. In *Proc. of IGCC*, 2011.

[66] R. Kaushik and M. Bhandarkar. Greenhdfs: Towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster. In *Proc.of USENIX*, 2010.

[67] R. Kaushik, M. Bhandarkar, and K. Nahrstedt. Evaluation and analysis of greenhdfs: A self-adaptive, energy-conserving variant of the hadoop distributed file system. In *Proc. of CloudCom*, 2010.

[68] M. Kendall and J. Ord. *Time-series*, volume 296. Edward Arnold London, 1990.

[69] E. Krevat, V. Vasudevan, A. Phanishayee, D. Andersen, G. Ganger, G. Gibson, and S. Seshan. On application-level approaches to avoiding tcp throughput collapse in cluster-based storage systems. In *Proc. of SC*, 2007.

[70] P. Suresh Kumar, P. Sateesh Kumar, and S. Ramachandram. Recent Trust Models In Grid. *JATIT*, 2011.

[71] N. Laoutaris and P. Rodriguez. Good things come to those who (can) wait or how to handle delay tolerant traffic and make peace on the internet. In *Proc. of HotNets-VII*, 2008.

[72] N. Laoutaris, G. Smaragdakis, P. Rodriguez, and R. Sundaram. Delay tolerant bulk data transfers on the internet. In *Proc. of SIGMETRICS*, 2009.

[73] Y. Lee, K. Chen, H. Su, and C. Lei. Are all games equally cloud-gaming-friendly? An electromyographic approach. In *Proc. of NetGames*, 2012.

[74] J. Leverich and C. Kozyrakis. On the energy (in)efficiency of hadoop clusters. *Operating Systems Review*, 44(1):61–65, 2010.

[75] M. Lin, A. Wierman, L. Andrew, and E. Thereska. Dynamic right-sizing for power-proportional data centers. In *Proc. of INFOCOM*, 2011.

[76] J. Liu, F. Zhao, X. Liu, and W. He. Challenges towards elastic power management in internet data centers. In *Proc. of ICDCS*, 2009.

[77] J. Lucas and M. Saccucci. Exponentially weighted moving average control schemes: Properties and enhancements. *Technometrics*, 32(1):1–29, 1990.

[78] D. Nagle, D. Serenyi, and A. Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *Proc. of SC*, 2004.

[79] S. Nedevschi, J. Chandrashekar, J. Liu, B. Nordman, S. Ratnasamy, and N. Taf. Skilled in the art of being idle: Reducing energy waste in networked systems. In *Proc. of NSDI*, 2009.

[80] M. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69(2):026113, 2004.

[81] L. Nikolaos, S. Michael, X. Yang, and R. Pablo. Inter-datacenter bulk transfers with netstitcher. In *Proc. of SIGCOMM*, 2011.

[82] A. Ojala and P. Tyrvainen. Developing Cloud Business Models: A Case Study on Cloud Gaming. *IEEE Software*, 28(4):42–47, 2011.

[83] Version 2 Parallel Virtual File System. http://www.pvfs.org, [Accessed in Feb 2015].

[84] A. Patro, S. Rayanchu, M. Griepentrog, Y. Ma, and S. Banerjee. The anatomy of a large mobile massively multiplayer online game. *SIGCOMM Computer Communication Review*, 42(4):479–484, 2012.

[85] D. Pittman and C. GauthierDickey. Characterizing virtual populations in massively multiplayer online role-playing games. In *Advances in Multimedia Modeling*, pages 87–97. 2010.

[86] PlanetLab. http://www.planet-lab.org/, [Accessed in Feb 2015].

[87] K. Psounis, P. M. Fernandez, B. Prabhakar, and F. Papadopoulos. Systems with multiple servers under heavy-tailed workloads. *Performance Evaluation*, 2005.

[88] J. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. In *Proc. of SIGCOMM*, 2010.

[89] S. Rajani and T. Rajender. Literature review: Cloud computing-security issues, solution and technologie. *International Journal of Engineering Research*, 3(4):221–225, 2014.

[90] P. Resnick, K. Kuwabara, R. Zeckhauser, and E. Friedman. Reputation systems. *Communications of the ACM*, 43(12):45–48, 2000.

[91] Amazon S3. http://aws.amazon.com/s3/, [Accessed in Feb 2015].

[92] P. Salvador and A. Nogueira. Study on geographical distribution and availability of bittorrent peers sharing video files. In *Proc. of ISCE*, 2008.

[93] H. Sawhney, A. Arpa, R. Kumar, S. Samarasekera, M. Aggarwal, S. Hsu, D. Nister, and K. Hanna. Video flashlights: real time rendering of multiple videos for immersive model visualization. In *ACM International Conference Proceeding Series*, volume 28, pages 157–168, 2002.

[94] H. Shen, Y. Lin, and J. Li. A social-network-aided efficient peer-to-peer live streaming system. *TON*, 23(3):987–1000, 2015.

[95] H. Shen and G. Liu. A lightweight and cooperative multi-factor considered file replication method in structured P2P systems. *TC*, 2012.

[96] H. Shen and C. Xu. Locality-aware and churn-resilient load balancing algorithms in structured peer-to-peer networks. *TPDS*, 18(6):849–862, 2007.

[97] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *Proc. of NSDI*, 2011.

[98] K. Shvachko, K. Hairong, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proc. of MSST*, 2010.

[99] The PeerSim simulator. http://peersim.sf.net, [Accessed in Feb 2015].

[100] J. Strauss, D. Katabi, and M. Kaashoek. A measurement study of available bandwidth estimation tools. In *Proc. of IMC*, 2003.

[101] R. Subrata and A. Y. Zomaya. Game-theoretic approach for load balancing in computational grids. *TPDS*, 19(1):66–76, 2008.

[102] Lustre File System. http://www.lustre.org, [Accessed in Feb 2015].

[103] ThinkServer. http://shop.lenovo.com/us/en/servers/, [Accessed in Feb 2015].

[104] R. Torres, A. Finamore, J. Kim, M. Mellia, M. Munafo, and S. Rao. Dissecting video server selection strategies in the youtube cdn. In *Proc. of ICDCS*, 2011.

[105] Sandia CTH trace data. http://www.cs.sandia.gov/Scalable_IO/SNL_Trace_Data/, [Accessed in Feb 2015].

[106] A. Verma, G. Dasgupta, T. Nayak, P. De, and R. Kothari. Server workload analysis for power minimization using consolidation. In *Proc. of USENIX*, 2009.

[107] H. Wang, H. Xie, L. Qiu, A. Silberschatz, and Y. Yang. Optimal isp subscription for internet multihoming: algorithm design and implication analysis. In *Proc. of INFOCOM*, 2005.

[108] S. Wang and S. Dey. Cloud mobile gaming: modeling and measuring user experience in mobile wireless networks. In *Proc. of SIGMOBILE*, 2012.

[109] Q. Wei, B. Veeravalli, B. Gong, L. Zeng, and D. Feng. Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster. In *Proc. of CLUSTER*, 2010.

[110] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variability: statistical analysis of ethernet lan traffic at the source level. *IEEE/ACM Trans. Network*, 1997.

[111] D. Wu, Y. Liu, and K. W. Ross. Modeling and Analysis of Multichannel P2P Live Video Systems. *TON*, 18(4):1248–1260, 2010.

[112] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. Madhyastha. Spanstore: cost-effective geo-replicated storage spanning multiple cloud services. In *Proc. of SOSP*, 2013.

[113] K. Xu, M. Zhang, J. Liu, Z. Qin, and M. Ye. Proxy caching for peer-to-peer live streaming. *Computer Networks*, 2010.

[114] Y. Yao, L. Huang, A. Sharma, L. Golubchik, and M. Neely. Data centers power reduction: A two time scale approach for delay tolerant workloads. In *Proc. of INFOCOM*, 2012.

[115] J. Zhang, F. Ren, L. Tang, and C. Lin. Modeling and Solving TCP Incast Problem in Data Center Networks. *TPDS*, 26(2):478–491, 2015.

[116] Z. Zhang, M. Zhang, A. Greenberg, Y. Hu, R. Mahajan, and B. Christian. Optimizing cost and performance in online service provider networks. In *Proc. of NSDI*, 2010.

[117] Z. Zhao, K. Hwang, and J. Villeta. Game cloud design with virtualized CPU/GPU servers and initial performance results. In *Proc. of ScienceCloud*, 2012.

[118] X. Zhuang, A. Bharambe, J. Pang, and S. Seshan. Player dynamics in massively multiplayer online games. *Carnegie Mellon University, Pittsburgh, Tech. Rep. CMU-CS-07-158*, 2007.